Bilkent University

Department of Computer Engineering

CS 319 Term Project

Section 2

Group 2G, Oldies but Goldies

Q-bitz

System Design Report

Project Group Members

1. Burak Kırımlı
2. Cansu Canan Ceyhan
3. Emre Keskin
4. Mert Çerçiler
5. Yağmur Özkök

Supervisor: Eray Tüzün

Table of Content

1. Introduction

1.1 Purpose of the System

Q-Bitz is a family board game. Our purpose is to implement Q-Bitz to computer in such a way that users can get the same and the highest satisfaction like the board game version. Our implementation is very similar with board game. In game there are 3 different game modes. First one is, user looks the design on the randomly opened card and tries to make the same design with using cubes. In the second game mode, user looks the design again and randomly rolls all cubes and tries to make the same design which is on the opened card. For the last game mode, user looks the design on the randomly opened card for 10 seconds and tries to make the same design with using cubes. Timing, memorizing and having fun is very important in our game.

1.2 Design Goals

The design step is very important about how the project will occur. In this step, we are going to analyze the project's needs and try to complete. To achieve that, we are going to use the non-functional requirements which we mentioned in the analysis report. In non-functional requirements, there are several subheadings which are usability, reliability, performance and supportability.

1.2.1 Usability

The most important requirement of usability is how easily a project (game) can be understood and used by the user. If a user has difficulty in understanding the game, he / she will not enjoy the game and the game loses its usability. In our project, the user interface is easily understandable. Switching between screens is set to be uncomplicated. The user can easily understand the game and play with fun. The user can play the game easily even without looking at "how to play".

1.2.2 Reliability

To prevent the possible crashes and bugs that can be occurred in the future "Q-Bitz" is going to be tested many times before released. All possible cases are going to be tested to release a reliable game. This property is in the reliability part of design goals for "Q-Bitz".

1.2.3 Performance

Performance is very important criteria for every project. In “Q-Bitz”, GUI library is going to be used for better performance and subsystems will be designed carefully to not to affect the performance in a bad way. In “Q-Bitz”, all user inputs are going to be acknowledged at most in one second. In our game, there is special peak time case when card is uploading and map is resetting. This case is special peak time for “Q-Bitz” and it is taking at most one second. This property is in the performance part of design goals for “Q-Bitz”.

1.2.4 Supportability

In supportability, there are two criteria which are adaptability and maintainability. For adaptability, “Q-Bitz”, will be designed to keep up with changes in Windows and Mac OS. For maintainability, since “Q-Bitz” is implemented in an object-oriented way, in future its unwanted features can be easily changed and new features can be added easily. These properties are in the supportability part of design goals for “Q-Bitz”.

2. High-Level Software Architecture

2.1 Subsystem Decomposition

In this section, we will decompose our system into subsystems, that means the identification of subsystems, services, and their associations to each other to easier to conceive, understand, program and maintain. By decomposing our system, we can modify or extend our game easier.

We have decided to use MVC (Model-View-Controller) architectural pattern while decompose our system. By using Model-View-Controller architectural pattern, we divide our system into three subsystems, our user interface classes for view part, game classes for model part, and controller and manager classes for controller part, as shown in the Figure 1. In our user interface classes, meaning classes that have menu, is designed like user chooses options before the game starts such as choosing game mode or difficulty level, as shown in Figure 3. Our view subsystem is responsible for application domain knowledge. In game classes, our game objects will

be created and games will be played in those sections, as shown in Figure 4. Our game subsystem is responsible for displaying application domain objects to user. Lastly, in our controller classes, control classes and manager classes will be implemented. Control classes will check to whether the players move accurate or not, or controls game is finished or not. Also, manager classes will manage the game play, as shown in Figure 5. Our controller subsystem is responsible for sequence of interactions with the user and notifying views of changes in the model.

As a result, our system decomposition provides us high cohesion and easier managing of our classes.

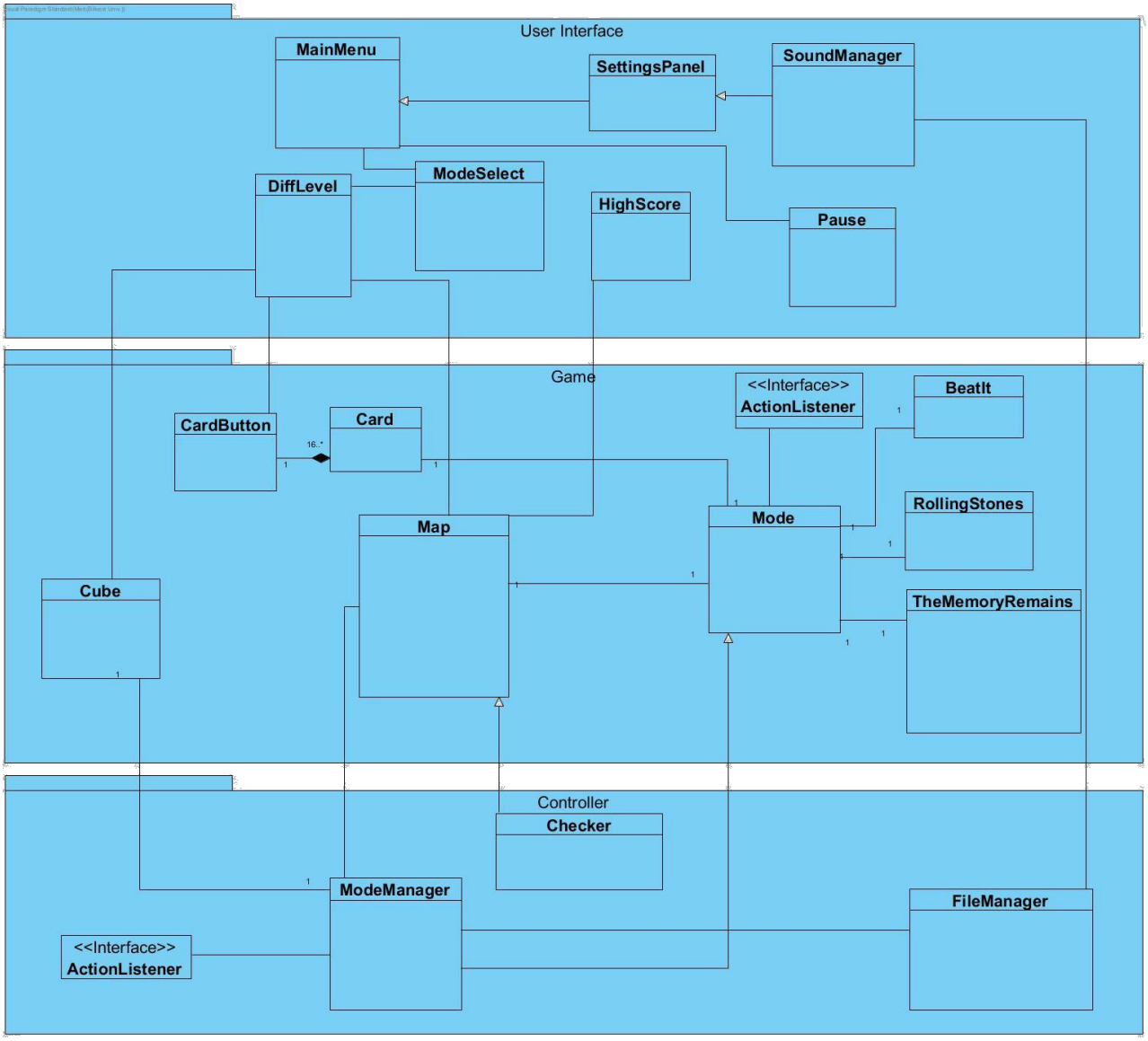


Figure 1: Basic Subsystem Composition

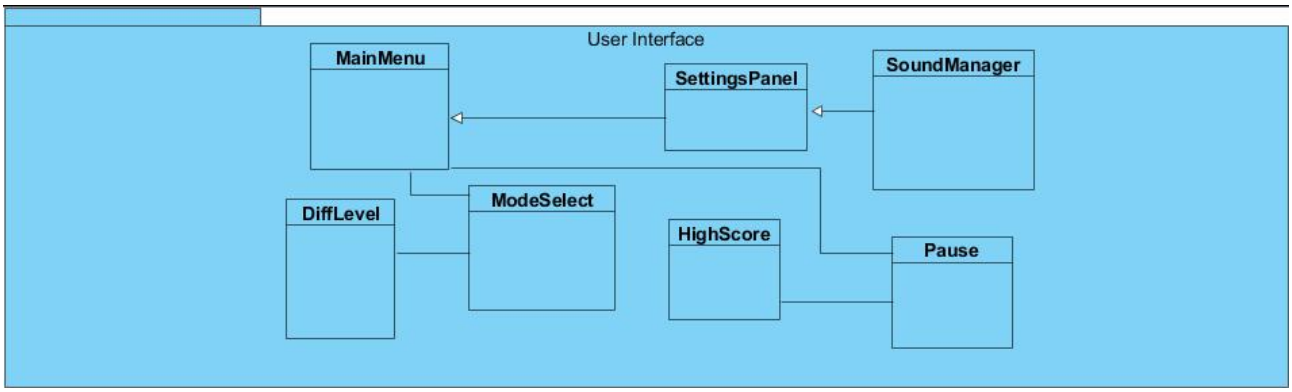


Figure 2: View subsystem (user interface)

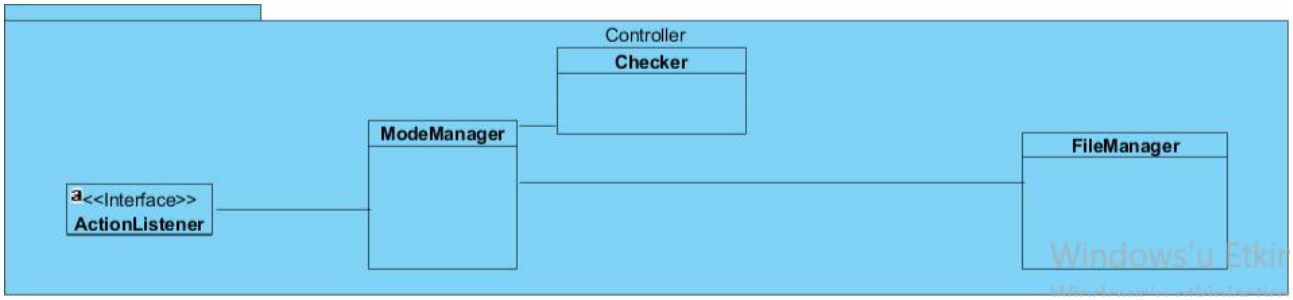


Figure 3: Controller subsystem

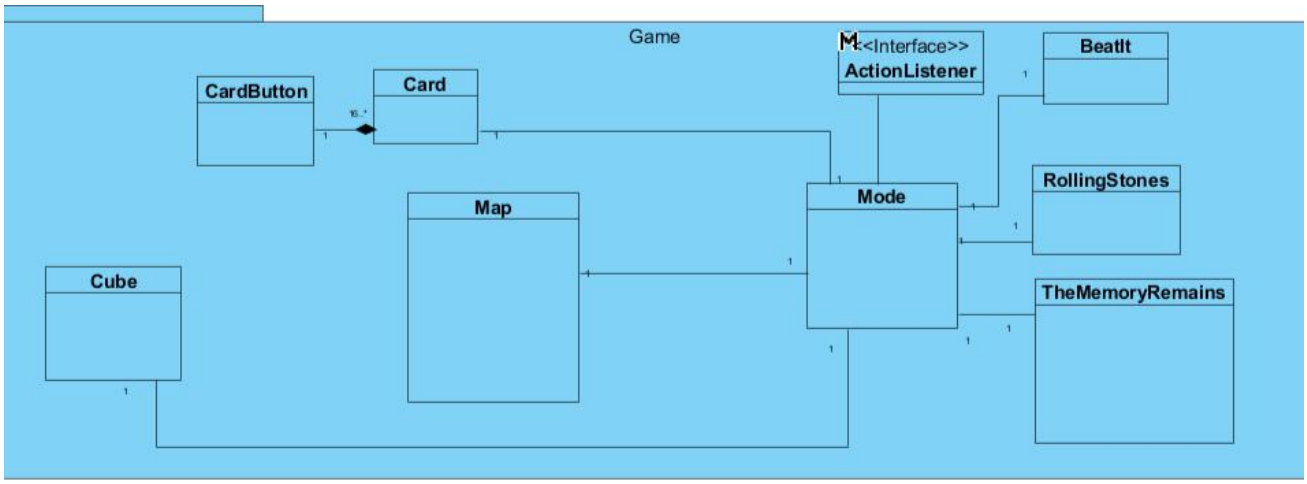
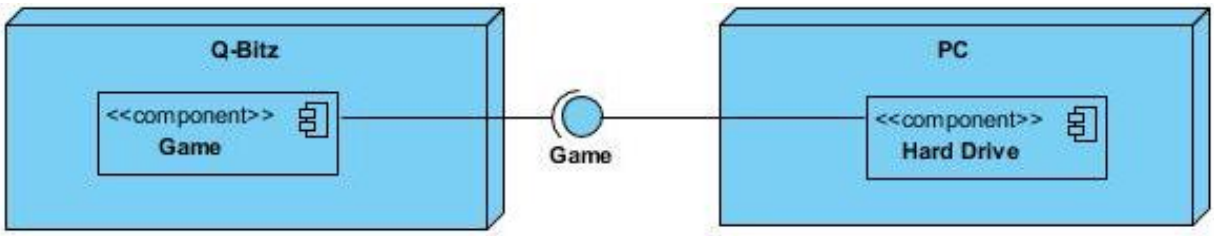


Figure 4: Model subsystem (Game)

2.2 Hardware/Software Mapping

As a group, we decide to implement our Q-bitz project in Java programming language. Graphical user interface (view part) of the Project will be implemented using JAVAFX libraries. Therefore, our software requirement is Java Development Kit(JDK) 8 or any newer version of JDK, so that our used libraries can be executed properly. Moreover, game will run in virtually all environments Linux, Mac and Windows which has Java Runtime Environment(JRA). Only hardware requirement of our Q-bitz is mouse. With usage of mouse, user can choose difficulty, game mode and which action he/she desires to do in actual gameplay. We intend to keep requirements as minimal as possible and hopefully any computer with a mouse and required JDK installed can run the game successfully. Q bitz will not require any internet connection or database, it will store everything needed in hard drive. Storage of high score will be done in a text file, images and sounds will be kept in the same folder as projects source code so they can be accessed easily.

2.2.1 Deployment Diagram



2.3 Persistent Data Management

In order to run properly, Q-bitz needs to store some data, yet it will not require any internet connection or data base operation system as discussed above, it will manage data in hard drive of the user. Objects which are crucial to the game such as map, cubes will be stored during run time phase and they are not modifiable. Moreover, in the same document as the game source code, there will be another file to handle image and sound. Images will be used for the puzzle that will be represented to the user, face of the cubes so that user can choose another cube by looking at the top of the cube. Also, according to user's game performance, we intend to display certain gifs, for instance if user puts 3 accurate cubes in a row without any errors there will be a certain gif. Same will be valid for certain numbers which are yet to be determined. When user is in main menu and he/she clicks to see the settings panel, user has the ability to change music that is

playing in the background. Sound options that we provide to the user will be in the same folder as the Images. Moreover, images will be stored using .jpg format and sound will be kept as .waw format.

2.4 Access Control and Security

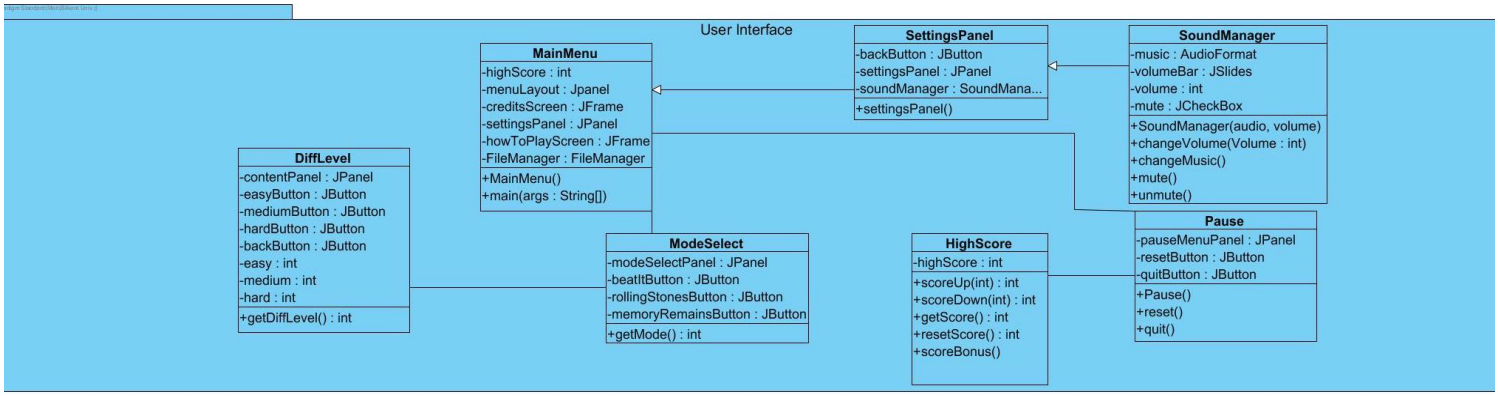
As discussed in the sections Hardware/Software Mapping and Persistent Data Management, our Project will not require internet connection nor database, therefore there will be no restriction or control checking for access. It makes sense considering there will not be any attributes user would like to store. There is only high score and it shows users high score starting from initial running of the program. So, every user will have unique high score when he/she runs the program. Considering there is no user profile, security is not an issue we need to consider in our game. Therefore, access control and security are not necessary for our program.

2.5 Boundary Conditions

Q-bitz will not require any install or download, in the final stage, it will be in an executable .jar and anyone who has that folder can run our project. Q-bitz can be terminated just by clicking the “X” button on the top left corner. It does not matter if the user presses that button in main menu, pause screen or during the gameplay, game will be terminated. If the program encounters any problems during uploading sounds or images which are part of the game, it will not crash or exit because such exceptions are handled in code. It will work without any visual or sound yet gameplay and other functionalities will not be affected. Moreover, if the game stops executing during gameplay due to performance issues any data will be lost.

3. Subsystem Services

3.1 User Interface Subsystem



User interface subsystem has 7 classes. They are all menus that user chooses options such as game mode or difficulty level. There are also Pause class, which user can either reset the game or quit the game. Settings class, that user change sound settings in it. Also, MainMenuPanel provides panel for the MainMenu class, which user has 4 options. (Play Game, Credits, HowToPlay and Settings)

3.1.1 MainMenu

Attributes:

- **private int highScore:** This will be used to show the high score in the game.
- **private JPanel menuLayout:** This will provide main menu layout via JPanel.
- **private JPanel settingsPanel**: This will provide settings screen layout via JPanel.
- **private viewFrame creditsScreen:** This will create the frame for credits screen.
- **private viewFrame howToPlayScreen:** This will create the frame for how to play screen.

- **private FileManager fileManager:** This attribute will be used in order to call the fileManager from FileManager class.

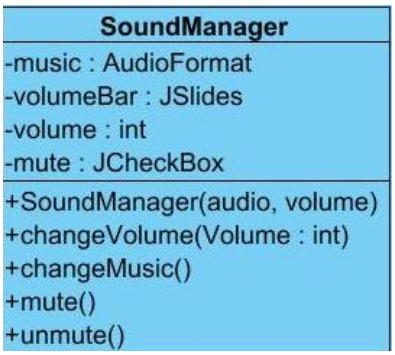
Constuctor:

- **public MainMenu():** Default constructor for the MainMenu class.

Mehtods:

- **public void main(args: String[]):** main method of the MainMenu class.

3.1.2 SoundManager



Attributes:

- **private AudioFormat music:** This will provide music of the game which user can change if he/she desires.
- **private JSlider volumeBar:** This will enable user to adjust the volume of the music playing in the background of the game.
- **private JCheckBox mute:** This will enable the option to mute the music.
- **private int volume:** This will hold the current volume value for the music.

Constructor:

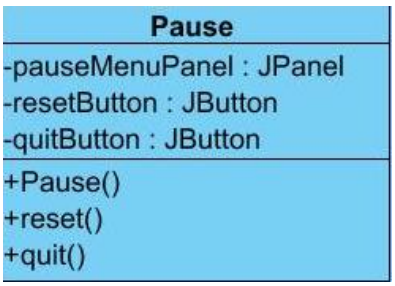
- **public SoundManager(audio: AudioFormat, volume: int):** Chooses default music and default volume.

Methods:

- **public void changeVolume(volume: int):** This will be used when user is adjusting volume using volumeBar slider stated above.

- **public void changeMusic():** This will choose and play another music from already defined options.
- **public void mute():** This will be used for muting the music.
- **public void unmute():** This will be used for unmuting the music.

3.1.3 Pause



Attributes:

- **private JPanel pauseMenuPanel:** This attribute will create panel for pause menu
- **private JButton resetButton:** Reset button for reset the game
- **private JButton quitButton:** Quit button for quit the game.

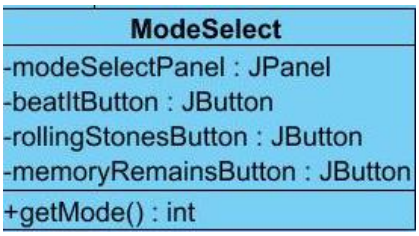
Constructor:

- **Pause():** constructor of pause

Methods:

- **public void reset():** method for resetting the game.
- **Public void quit():** method for quitting the game.

3.1.4 ModeSelect



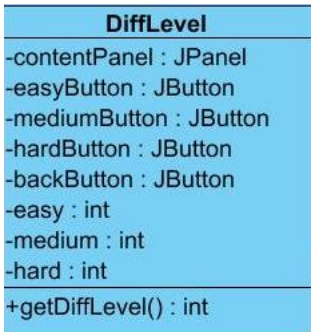
Attributes:

- **private JPanel modeSelectPanel:** This attribute creates panel for mode select menu.
- **private JButton beatItButton:** beatItButton for the Beat It mode.
- **private JButton rollingStonesButton:** rollingStonesButton for the Rolling Stones mode.
- **private JButton memoryRemainsButton**: memoryRemainsButton for the Memory Remains mode.

Methods:

- **public int getMode():** getter method of ModeSelect class. Returns selected mode.

3.1.5 DiffLevel



Attributes:

- **private JPanel contentPanel:** This attribute will be used for creating “Content” panel.
- **private JButton easyButton:** This attribute will be used for creating “easy” button to let the user select the easy game mode.
- **private JButton mediumButton:** This attribute will be used for creating “medium” button to let the user select the medium game mode.
- **private JButton hardButton:** This attribute will be used for creating “hard” button to let the user select the hard game mode.
- **private JButton backButton:** This attribute will be used for creating “back” button to let the user go back the previous page.
- **private int easy:** This attribute will be used to represent the easy game mode with using “int” type.

- **private int medium:** This attribute will be used to represent the medium game mode with using “int” type.
- **private int hard:** This attribute will be used to represent the hard game mode with using “int” type.

Methods:

- **public int getDiffLevel():** Method to get the user’s difficulty level selection.

3.1.6 Settings

Attributes:

- **private JButton backButton:** This attribute will be used for creating “back” button to let the user go back the previous page.
- **private JPanel settingsPanel:** This attribute will be used for creating JPanel for “Settings” menu.
- **private SoundManager soundManager:** This attribute will be used to call soundManager object from SoundManager class. In “Settings”, user can select the background music.

Constructor:

- **public settingsPanel():**default constructor for the settingsPanel.

3.1.7 HighScore

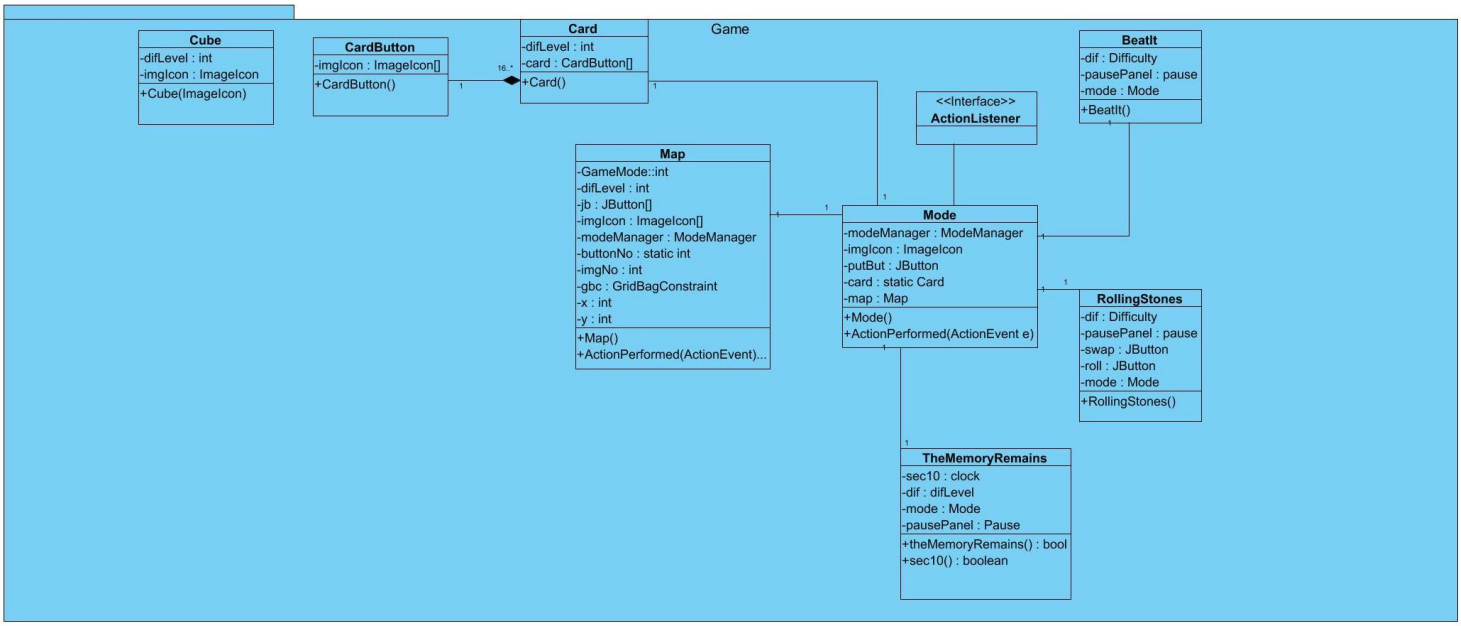
Attributes:

- **private int highScore:** This attribute is going to be used for the keep track of the high score of the user.

Methods:

- **public void scoreUp(int):** This method is going to be used to increase the high score.
- **public void scoreDown(int):** This method is going to be used to decrease the high score.
- **public int getScore():** This method is going to be used for getting the high score.
- **public void resetScore():** This method is going to be used for resetting the high score.
- **public void scoreBonus():** This method is going to be used for adding bonus point to high score when special conditions satisfied.

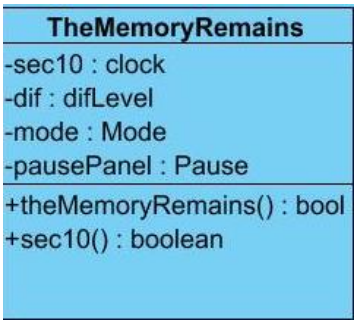
3.2 Game Subsystem



Game subsystem is the subsystem where game objects are created. We have 3 Game Objects which are Cube, Card and Map. Mode class is creating the game by doing put the commands and game objects into the panel. Also, in the game subsystem, we have game

mode classes, which are theMemoryRemains, RollingStones and BeatIt. Player will play the game mode which he/she selected earlier. Lastly, the highscore class that hold the score of the player and the highscore.

3.2.1 TheMemoryRemains



Attributes:

- **private Clock sec10:** This will be used for 10 sec time limit user has for seeing the puzzle in this mode.
- **private Mode mode:** This will be get commands like put and change buttons, game objects like maps, cubes and card, and general rules like time checker.
- **private DiffLevel dif:** This will hold the current difficulty level.
- **private Pause pausePanel:** This will be used for pausing the game in this mode.

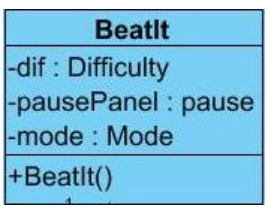
Constructor:

- **public TheMemoryRemains ():** Default constructor for the TheMemoryRemains class.

Methods:

- **public boolean sec10(volume: int):** This will be used at the start of the game, it will return false after 10 seconds and user will not be able to see the puzzle anymore.

3.2.2 BeatIt



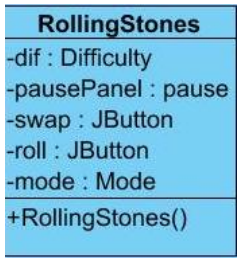
Attributes:

- **private int diff:** This attribute will be used to get selected difficulty level.
- **private Pause pausePanel:** This attribute will be used to call the pausePanel object from pause class.
- **private Mode mode:** This will be get commands like put and change buttons, game objects like maps, cubes and card, and general rules like time checker.

Constructor:

- **public BeatIt ():** Default constructor for the BeatIt class.

3.2.3 Rolling Stones



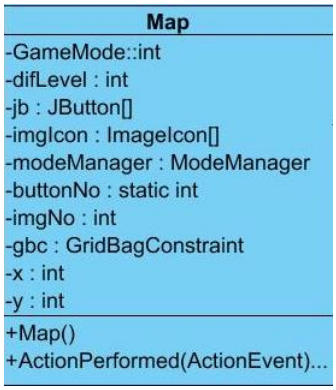
Attributes:

- **private Difficulty dif:** This attribute will be used to reach to get the size of the map. As maps size is related to difficulty.
- **private Pause pausePanel:** This attribute brings pause button with it's attribute.
- **private Mode mode:** Mode brings methods for roll put swap and roll.
- **private JButton swap:** Creates canvas for swap method from ModeManager class.
- **private JButton roll:** Creates canvas for roll method from ModeManager class.

Methods:

- **public RollingStones():** Creates a constructor to recall in main JFrame.

3.2.4 Map



Attributes:

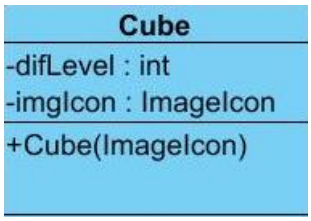
- **private int gameMode:** This attribute will be used to get the selected game mode (BeatIt, Rolling Stones, Memory Remains)
- **private int difLevel:** This attribute will be used to get the selected difficulty level.
- **JButton jb[]:** This attribute will create the map buttons for the map. Map consists of buttons.
- **ImageIcon img[]:** Image array is used to insert images that player select into the map.
- **ModeManager modeManager:** This object is used to get imgNumber attribute in the mode manager class.
- **static int buttonNo:** This attribute holds the button number that user selected in the map.
- **int imgNo:** This attribute takes the image number that button holds in the map.
- **GridBagConstraints gbc:** GridBagConstrains makes the map buttons grid.
- **int x:** This attribute is used for x coordinate of the buttons
- **int y:** This attribute is used for y coordinate of the buttons

Constructor:

- **public Map():** This method will be used to create map Methods:
- **public void ActionPerformed(ActionEvent e):** This method is taken from the ActionListener interface. This method has several works. First, whenever user clicks the map button, image of the button changes according to image of the cube, if put, swap or roll buttons did not click before clicking map button. Also, put button, swap

button, and roll buttons are implemented in this method because these instructions are instructions are done by clicking the map buttons in the map.

3.2.5 Cube



Attributes:

- **private int difLevel:** This attribute gets selected difficulty level. Cubes are created according to selected difficulty level.
- **ImageIcon imgIcon[]:** ImageIcon array takes images for the cube.

Constructor:

- **Cube(ImageIcon img):** Default constructor of the Cube class. Cube's image is decided according to ImageIcon parameter.

3.2.6 CardButton

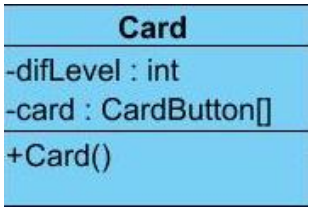
Attributes:

- **ImageIcon imgIcon[]:** ImageIcon array is used to insert images into the each card label.

Constructor:

- **CardButton():** In constructor, all imgIcon instances are initialized and CardButton is created. CardButtons creates card.

3.2.7 Card



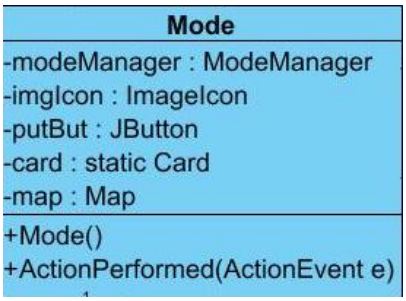
Attributes:

- **private int difLevel:** This attribute will be used to represent difficulty level which is selected by user(easy, medium, hard). Number of CardButton object is determined according to difficulty level.
- **CardButton[]** card**:** This attribute takes the array of CardButton objects. Each object is a single label of cards.

Constructor:

- **public Card():** default constructor for the Card. Card is created in the constructor.

3.2.8 Mode



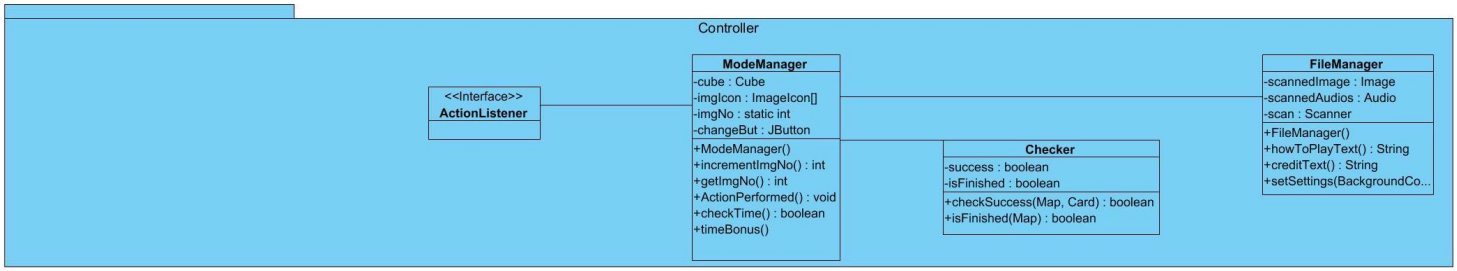
Attributes:

- **private ModeManager modeManager:** This attribute gets instance of Mode Manager class to hold commands and timeChecker.
- **ImageIcon imgIcon:** imgIcon gets the image of the first side of the cube, for default image at the beginning of the game.
- **JButton putBut:** JButton for the put button.
- **static Card card:** Creates instance of a card. It is static because this card object has to be reachable from Checker class.
- **Map map:** Creates instance of a map.

Methods:

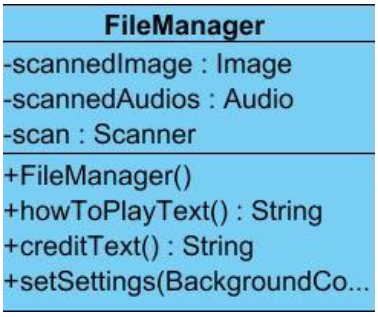
- **public Mode():** This method creates game. Put maps, cube and card commands and timeChecker.
- **public void ActionPerformed(ActionEvent e):** This method is taken from the ActionListener interface. This method focuses the clickable map buttons in the map. It makes the border of the clickable buttons green so that user can understand which buttons are clickable.

3.3 Controller Subsystem



In our controller subsystem, we have 4 classes. Mode manager is the class where game commands will be implemented. Our commands are change (for the Memory Remains and BeatIt game modes), swap and roll (for the Rolling Stones game mode), and change (common for all game modes). Also, time checker will be implemented in this class, which will check whether time is up or not. Checker class will control the user actions, whether he or she put the cube correctly or not. isFinished class will control whether the map is finished or not. Lastly, FileManager class will scan images (for the cubes and cards) and audios (for background music).

3.3.1 FileManager



Attributes:

- **private Images scannedImages:** This attribute will be used to scan images, that are going to be used in entire game.
- **private Audio sacnnedAudios:** This attribute will be used to scan audios, that are going to be used in entire game.
- **private Scanner scan:** This attribute will be used to call the scan object from.

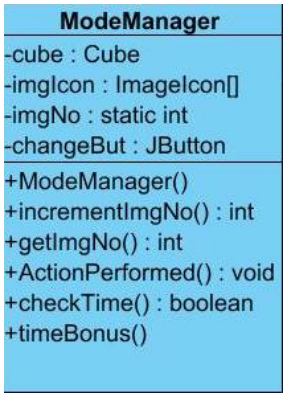
Constructor:

- **public FileManager():** default constructor of the File Manager.

Methods:

- **public String howToPlayText():** This method will return the text for “How to Play” section.
- **public String creditsText():** This method will return the text for “Credits” section.
- **public void setSettings(colorArray, audioArray):** method to set background color and background audio for the game.

3.3.2 ModeManager



Attributes:

- **ImageIcon imgIcon[]:** This attribute gets image array for the implementation of change button. Change button changes images from the imgIcon array.
- **JButton changeBut:** Creates JButton for the change button.
- **static int imgNo:** This attributes holds the image number. It changes when cubes image is changed. It is static because it has to be called from the Map class.

- **Cube cube:** This object gets the Cube object, when cubes image is change, another Cube object is created with the parameter ImageIcon.

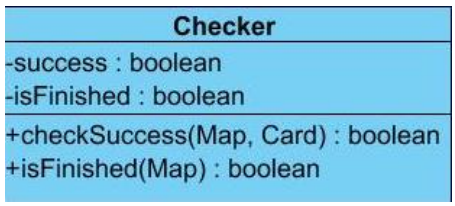
Constructor:

- **ModeManager():** This is a constructor for the ModeManager class.

Methods:

- **public int incrementImgNo():** This method increments the imgNo attribute when change button is clicked.
- **public int getImgNo():** This is a getter method of the imgNo.
- **public void actionPerformed(ActionEvent e) :** This method is taken from the ActionListener interface. This method contains implementation of the change button.
- **public boolean checkTime():** This method checks if time is up or not. Related to this game will continue or will end.
- **public void timeBonus(Map):** This method freezes time for specific time period when user places the cube correctly specified area of the map.

3.3.3 Checker



Attributes:

- **boolean success:** It takes true when user placed cube into the map correctly, and otherwise it takes false.
- **boolean isFinished:** It takes true when user finished the map, and otherwise it takes false.

Methods:

- **public boolean checkSucccess(Map, Card):** Checks whether the cube is placed correctly or not into the map.

- **public boolean isFinished(Map):** Checks whether the map is finished or not. If map is finished it returns true, and otherwise false.

4. Low-level Design

4.1 Object design trade-offs

4.1.1 Fault tolerance vs Functionality

In our project, our aim is to have a fault tolerance game. When user does any action which is not mentioned in activity diagram, our game must tolerate that action rather than stop that game and come across with exceptions. In order to keep fault tolerance, functionality is limited. As there are more functions, there are more activities and more wrong cases to be tolerated. We limit functionality by using buttons. Rather than giving unlimited functions to user, we specify the functions of buttons and user can use only functions of those buttons. So we limit functionality in order to keep fault tolerance.

4.1.2 Robustness vs High-Performance

In our project as our developers are more experienced in java, java language is used. So a game written in java will be more robust as developers are more experienced. So robustness of the game maintained. On the other hand, because of the memory allocation c++ provides high speed performance. To keep robustness instead of high performance a language selected based on experience of developers instead of languages performance.

4.1.3 Maintainability vs Functionality

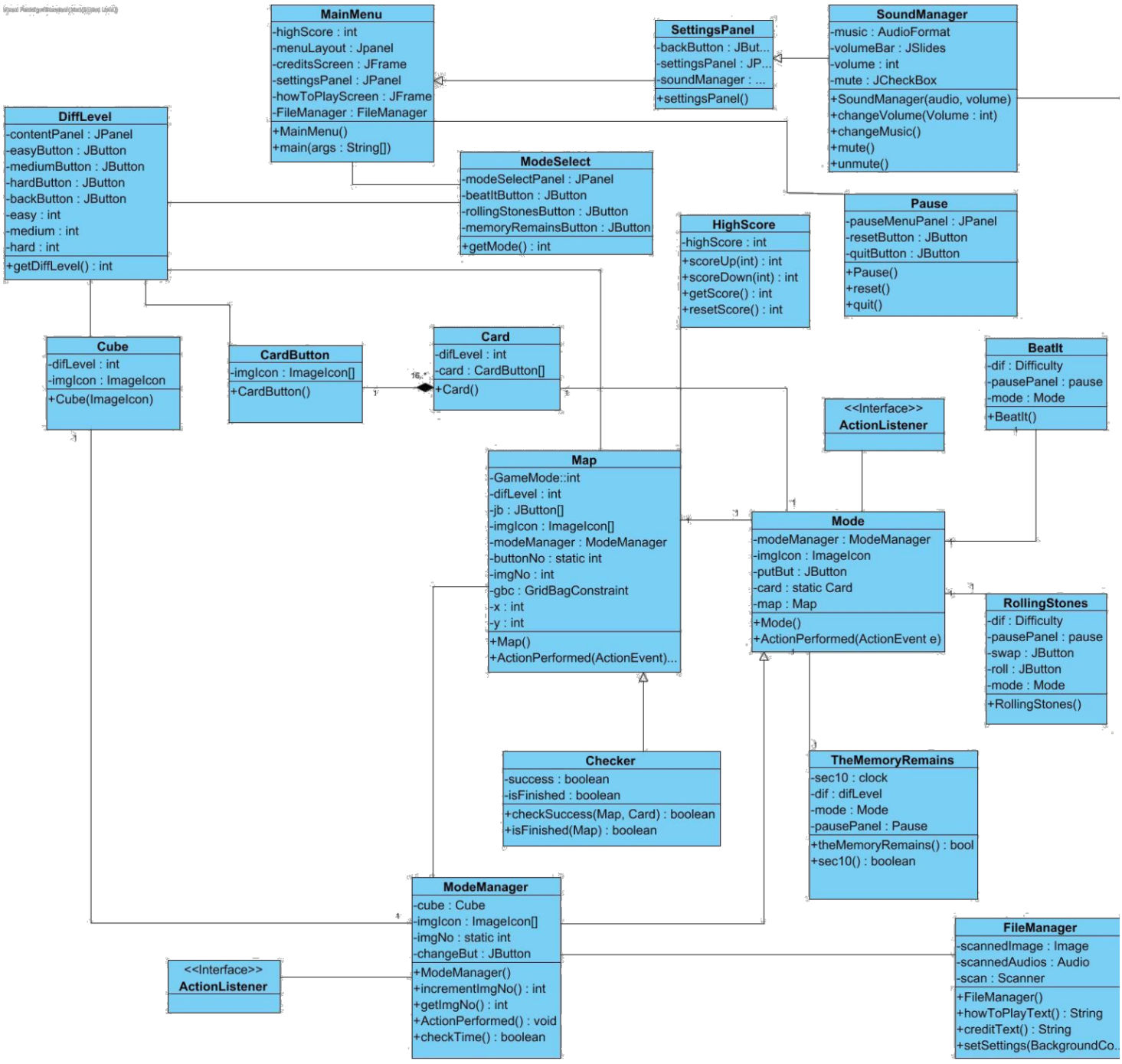
Maintainability is based on correcting faults and modifying the software. In our game to keep the game maintainable, game functions limited. As the game is more simple it is more easy to keep the maintenance. When functions get simple, it is more easy to modify to correct faults and improve performance and attributes. Also adapting the game to different environments get easier. So in order to keep maintainability, we trade-off functionality.

4.1.4 Rapid Development vs Adaptability

In order to rapid development, we limited the adaptability. As adaptability is based on the system adaption to interactive system between user and system. Adaption is limited, by limiting user's actions in interactive system. Users actions limited by functionalities of the

game. Limitation of adaption ease the rapid development. So we trade-off adaption vs rapid development.

4.2 Final Object Design



4.3 Packages

4.3.1 java.util

This package contains concurrent which contains time unit methods.We will use these package for checking time.

4.3.2 java.awt.gridLayout

We will use this layout for maps to place the cubes.

4.3.3 java.awt.actionEvent

Implements ActionListener interface for events.

4.3.4 javax.swing.*

For using JLabels, JButtons, JFrames etc. we need this package. Because our game is highly visual all packages in swing is useful.

4.3.5 javafx.scene.image

This package is useful for importing pictures. As we use figures on the cubes and a picture on the card this package is essential.

4.4 Class Interfaces

4.4.1. ActionListener

Especially, as shown in our mock up, there are too many buttons used in the game. For adding function to those buttons, action listener interface used.

4.4.2. MouseListener

We will use mouse listener for selecting cubes. As selected cubes will be used as parameters for functions. Mouse listener interface is also essential.

5. Improvement Summary

For the second iteration, we added following things:

- We improve the introduction.
- System architecture fixed.
    - a. Hardware- software mapping fixed.
    - b. Subsystem decomposition explained.
    - c. New features added to persistent data management.
- Design goals subheadings changed.
    - a. Extendibility is deleted instead reliability added.
    - b. Portability is deleted instead supportability added.
    - c. Usability
    - d. Performance
- New class added called CardButton.
- ActionListener gadded to class diagram.
- scoreBonus() method added to HighScore class and timeBonus() method added to ModManager class.
- Class links fixed.
- Deployment Diagram added.
- New subheadings added to trade-off