



# **IQ Puzzler**

## **Design Report**

### ***Game Of Objects***

---

**Zafer Çınar  
Arda Türkoğlu  
Engin Deniz Kopan  
Mehmet Selim Özcan  
Mehmet Sanisoğlu**

**Nov 22, 2018**

## ***1.Introduction***

IQ Puzzler is a board game that features different kind of shapes aiming to be placed on a 5 to 11 grid while aiming to fill them all by using all of the given shapes. In every new game certain shape are placed beforehand to restrict player placements and thus create challenges for the player.

### ***1.1 Purpose of the System***

IQ Puzzler is a 2D, and potentially 3D, puzzle board game. The game is designed with great care so that the virtual version will include all the features of the physical game itself with potential to include several more add-ons. The game is over when there is no empty space left on the game board, which is possible if and only if all the pieces are in their correct places, excluding the pre-placed pieces. It includes several different scenarios to challenge the players' minds. Game is aimed to be challenging, user-friendly, fluent and with moderate graphics.

### ***1.2 Design Goals***

A good implementation requires a good design, that's why in this design part we need to go more into detail and describe the parts in our non-functional requirements explicitly.

#### ***1.2.1 User Requirements***

These are the aspects that the user is mainly concerned about. It includes the parts that make up a playable game.

##### ***1.2.1.1 Performance***

In order to make the virtual version as good looking as possible with smooth framerate expectations, we preferred to use JavaFX for our implementation process.

##### ***1.2.1.2 Ease of Use (Usability)***

Usability is one of the key factors in any project and even more so in ours. Most of the potential players will be learning this game for the first time and so should not be

bothered by the user interface. The user interface will be self-explanatory, simple and intuitive to grasp that will not create any problem for the user. Our main menu will be as simple as possible with instructions to select the game mode desired and acquire the settings preferences. Our system will be mainly based on mouse activities, so the user will not be bothered with any keyboard inputs or key bindings, except for the predefined pause “P” and menu “Esc” buttons.

### **1.2.2 System Requirements**

There are certain crucial aspects to consider for the developer side of the project. In order to present a solid work, we want the project to be easily extendable and reusable in other similar projects while ensuring its adaptability to various platforms.

#### **1.2.2.1 Extendibility**

Our design lets us to modify our game with ease. Although the gameboard map and shapes toolbar will stay the same, by changing the implementation and the rules of the game we aim to add several new game modes to our new system like “Time Bomb” and “Rotating Map” game modes.

#### **1.2.2.2 Reusability**

The main skeleton of our program, which consists of a large display panel that can place and display the objects in the container next to it, can easily be used for another similar or entirely different project with various purposes.

#### **1.2.2.3 Multi-Platform**

We want our system not to restrict itself with only one platform but rather be accessible to various platforms. That’s why we preferred to use Java, which can run on several different systems and devices.

### 1.3 Definitions

MVC: Model View Controller

JavaFX: a software platform for creating and delivering desktop applications

JDK8: Java Development Kit 8

## 2. System Architecture

System architecture is crucial in the sense that it decides on the trade-off we get during the implementation. We aim to get the simplest solution possible while trying not to extend each class and increase the coupling and increase the complexity of the program as a whole.

### 2.1 Subsystem Decomposition

For our project we categorised our system into three main part: Visualization, Game Engine and Data Management. Each category can be easily modified within themselves and so any additions to the project will be fairly easy to implement and keep track of. In this sense, we preferred an MVC pattern to use in our system.

The View part will consist of the GUI component creators that display the menu, grid, shapes and other user interface elements. This part will be responsible for the visualization of the game itself and will be easily adaptable to any change in the model.

Controller will be the GameManagerer class which will basically be the game logic that runs the game logic and respond to the changes in the system. It will act as a coordinator between the game objects and the visual representations of them and keep the game logic up and running.

Lastly, the game objects as well as the game grid will be the Model part of the project that will be interacting heavily with each other with the help of controller and view parts.

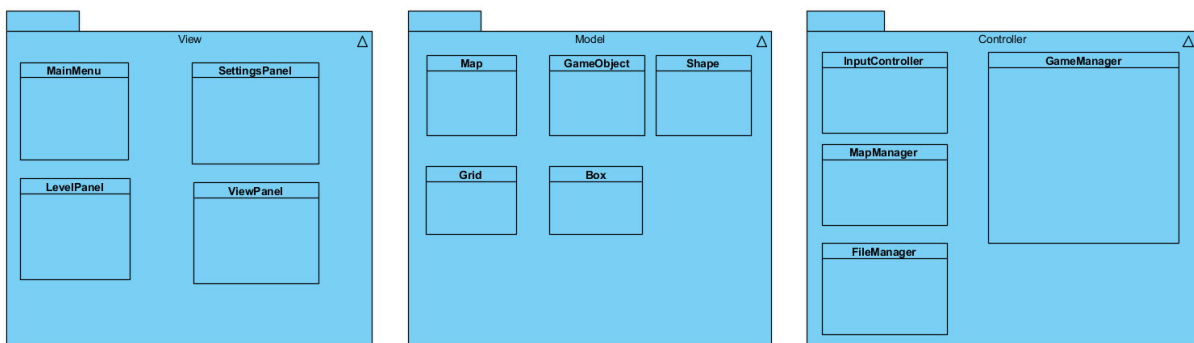


Figure 1- MVC Components of the system

The view part consists of MainMenu, SettingsPanel, LevelPanel and ViewPanel. MainMenu will act as a user interface that is showed at the game start. The mouse event on the MainMenu will take the user to another panel according to their choice. The view part provides the user with a visual guide in game by using the GameManager instance of the game in the controller part. By using the user interface elements such as SettingsPanel and LevelPanel, the player interacts with the GameManager and customizes the game, which in turn presents the desired game mode to the player.

The model part includes all the object classes related to game pieces as well as the Map with Grid. The model part provides other View and Controller with the necessary objects that are heavily used and interacted with each other. GameObjects are created in GameManager of the Controller part as an essential part of the game logic and they are represented in the View part visually, similar to the other object elements in the system.

The controller acts as the game logic in the system and manages the interactions between the objects. It requires the game objects in the model part and implements the game logic according to the interactions that the user executes and provides a response. This response is then seen as a visual result in the View part.

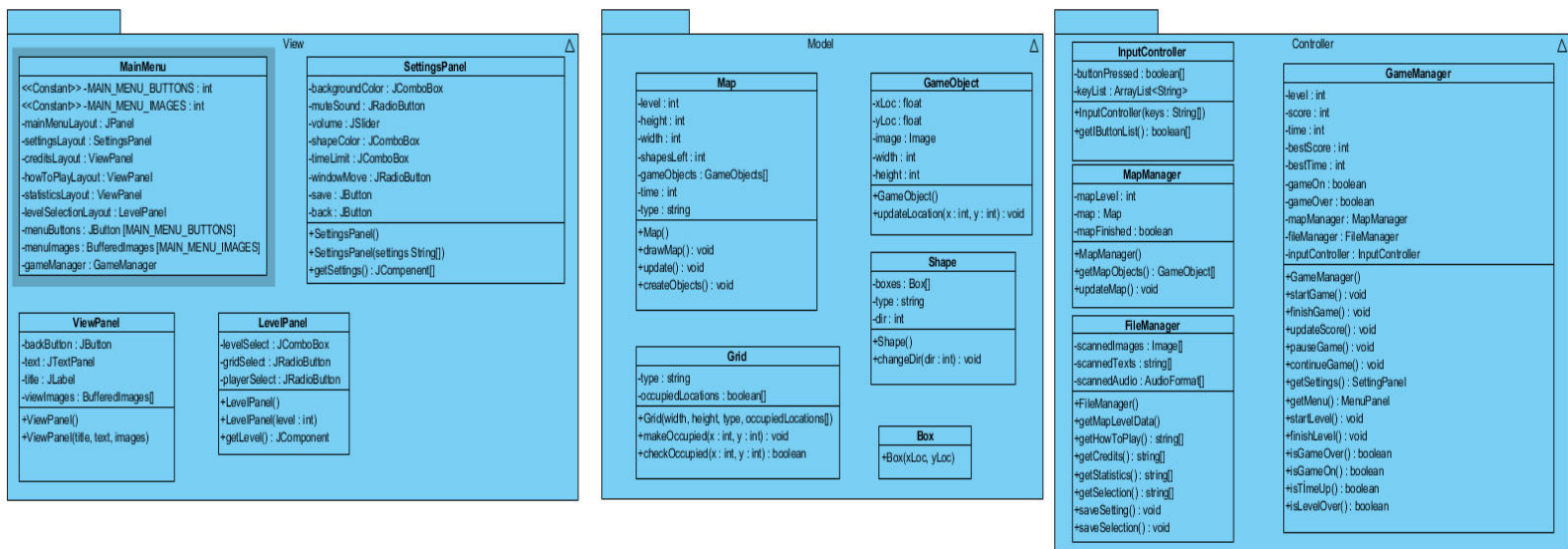


Figure 2- MVC class diagrams

## ***2.2 Hardware / Software Mapping***

IQ Puzzler will require a set up with Java Runtime Environment to run the standalone program. Also, since the graphical visualisation of the game requires JavaFx the setup will also require an updated Java Development Kit version, preferably JDK8 8 or higher.

Operating on a suitable setup with these conditions, the game requires certain hardware requirements to function. A keyboard is preferred for the mapping of certain shortcuts for the in-game actions such as quit, menu and pause. These actions will also be able to perform via using the mouse in the in-game user interface. Unlike a keyboard, a mouse is mandatory as most of the interactions are performed with the mouse clicks and movements.

## ***2.3 Persistent Data Management***

IQ Puzzler will not require database storage for it to function properly. The default configurations and object data are stored in the game folder in the hard drive of the user in .txt format. The game will be accessing these files to implement game logic. Game statistics and settings preferences of the user will be saved in these .txt files and the game will present a customized playthrough to the user.

## ***2.4 Access Control and Security***

The game will be operating according to the currently selected settings and present a game accordingly. There is no sensitive data or account present, so we decided not to implement any security measures to the game.

## ***2.5 Boundary Conditions***

IQ Puzzler will be in a compact form with a .jre extension. Since Java Runtime Environment files can run on almost every platform the game will not face problem when transported and shared across platforms. When the game starts, it will keep running so long as the user does not quit the game via the in-game menu.

### 3.Subsystem Services

We have 3 different subsystem in our design. All subsystems work each other in order to create working game.

#### 3.1 User Interface Subsystem

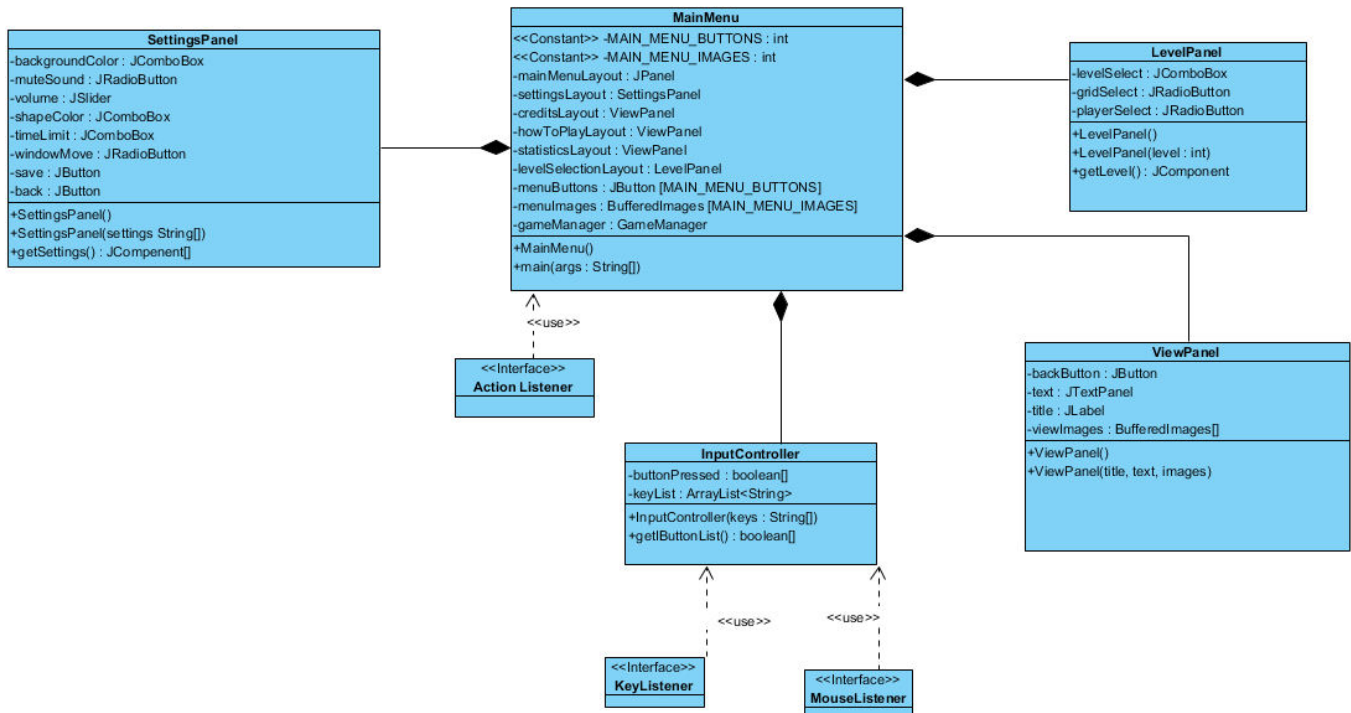


Figure 3 – User Interface Subsystem

User Interface Subsystem consists of seven classes and is responsible for the establishment of the interface between the system and the user. “MainMenu” class is the main class of the User Interface Subsystem. Actually it is not totally main design pattern however as a functionality relation the only change in Game Management subsystem should have happen through MainMenu class. It has an Action Listener to take events and check them for the game. InputController class works with GameManager system and have two interfaces called KeyListener and MouseListener. ViewPanel class helps to display the other classes mentioned in diagram. Level Panel opens a new panel to choose level. SettingsPanel class also open a panel for changing settings like background,sound , colors, time limit, save setting etc. MainMenu class is a main class, because all the classes are linked to it and the user can access them from this class. The user can also access “Play”,“How to Play”, “LevelSelection”,“Statistics”,“Credits” and “Quit” screens from here. The user will see 6 buttons on the Menu display; play, level select,how to play, settings,statistics, credits and

quit. Each screen will be initialized with the help of the instances of the classes mentioned before. For instance, by entering the Settings screen, the user will be able to adjust his/her settings with the help of buttons and buttonPressed from the InputController, which gets the boolean array of buttons pressed.

### 3.1.1 MainMenu Class

MainMenu
<pre> &lt;&lt;Constant&gt;&gt; -MAIN_MENU_BUTTONS : int &lt;&lt;Constant&gt;&gt; -MAIN_MENU_IMAGES : int -mainMenuLayout : JPanel -settingsLayout : SettingsPanel -creditsLayout : ViewPanel -howToPlayLayout : ViewPanel -statisticsLayout : ViewPanel -levelSelectionLayout : LevelPanel -menuButtons : JButton [MAIN_MENU_BUTTONS] -menuImages : BufferedImages [MAIN_MENU_IMAGES] -gameManager : GameManager +MainMenu() +main(args : String[]) </pre>

Figure 4 – Main Menu

“MainMenu” is the first class that will be instantiated when the game is first executed and displays the main menu. In this menu, there are six buttons name menuButtons such as “Play”, “How to Play”, “LevelSelection”, “Statistics”, “Credits” and “Quit”. These buttons will open panels for each button. MenuImages are images for buttons and gameManager is the object to check game. “MainMenu()” method works as a main method for the game. This class will call other classes and method in it.

### 3.1.2 SettingsPanel

SettingsPanel
<pre> -backgroundColor : JComboBox -muteSound : JRadioButton -volume : JSlider -shapeColor : JComboBox -timeLimit : JComboBox -windowMove : JRadioButton -save : JButton -back : JButton +SettingsPanel() +SettingsPanel(settings String[]) +getSettings() : JComponent[] </pre>

Figure 5 – Settings Panel



This panel is panel to change game settings. User can change attributes as background color , volume, sound on/off, shape color , time limit , save settings. Also, it has a back button to go back to menu. Color change and time limit attributes are JComboBox, other attributes are JradioButton.

### 3.1.3 LevelPanel

LevelPanel
-levelSelect : JComboBox -gridSelect : JRadioButton -playerSelect : JRadioButton
+LevelPanel() +LevelPanel(level : int) +getLevel() : JComponent

Figure 6 – Level Panel

LevelPanel class is opened after “Level Selection” button pressed in MainMenu. It has JComboBox called levelSelect and there are number of levels in it. User can select the grid and player with the radioButton. It has LevelPanel() method to create panel. getLevel() method returns the selected level.

### 3.1.4 ViewPanel

ViewPanel
-backButton : JButton -text : JTextPanel -title : JLabel -viewImages : BufferedImages[]
+ViewPanel() +ViewPanel(title, text, images)

Figure 7 – View Panel class

This class helps MainMenu class to create panels. It has title text and back button in it. Also, it can show images from the BufferedImages[] array.

### 3.1.5 Input Controller

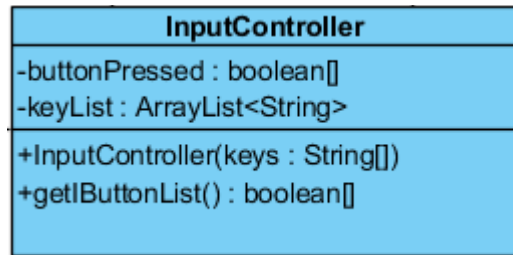


Figure 8 – Input Controller class

This class controls and gets the inputs from the buttons. This class has attributes named `buttonPressed` and `key list`. `buttonPressed` hold buttons pressed in a boolean array. `getButtonList()` method returns the `buttonPressed` array to the user.

### 3.2 GameManager Subsystem

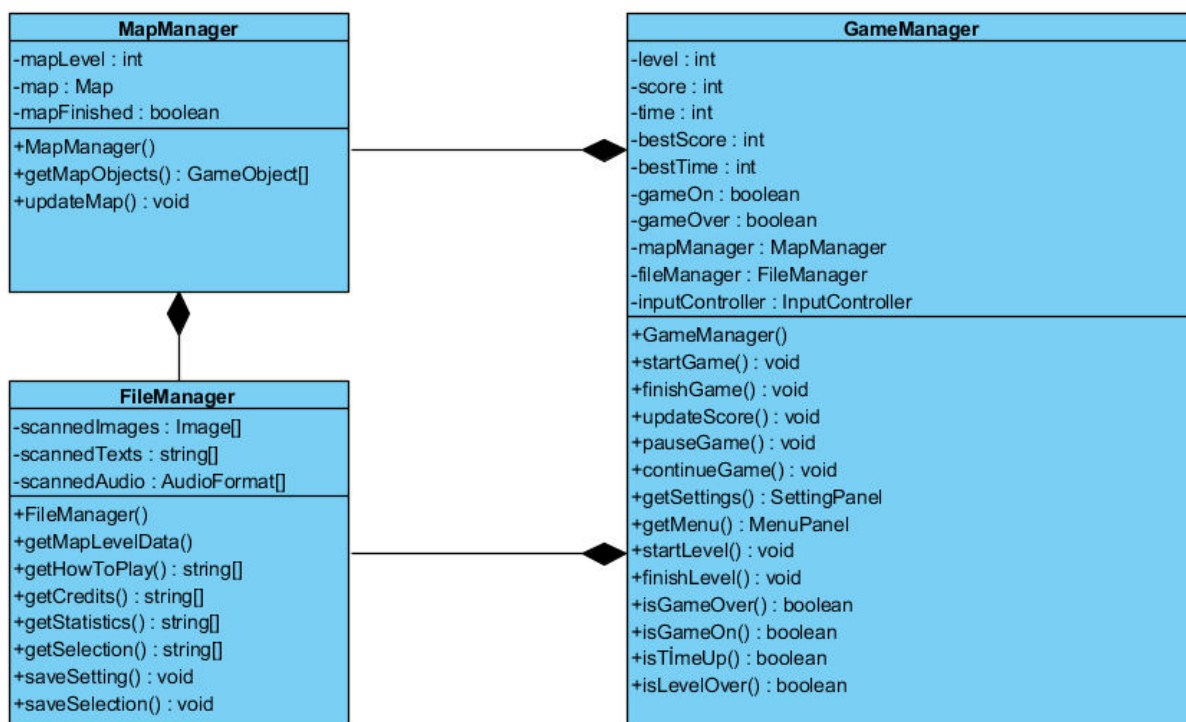


Figure 9 – Game Manager Subsystem

Game Manager System consists of three classes and is responsible for the game logic and handling the game mechanisms such as play and display. In this subsystem, **GameManager** is the main class which holds the game logic. So, it holds the instances of **MapManager**, **FileManager** classes in it. **GameManager** class reads the files via

FileManager and determines the level, the map of the level, the objects on the map and passes it to MapManager class. MapManager itself, manages the Map class, thus it is a main class, which controls the class under it. GameManager class is the primary class for this subsystem.

### 3.2.1 Game Manager



Figure 10 – Game Manager class

For this class definition, all of the system are going to be based on our game manager class. It is going to provide functionality for our sub-systems by interacting with map manager and file manager while a game is played. It plays a vital role in the maintenance of the game

since we need a base class which helps all other class to pull together so as to perform their tasks simultaneously.

In the game manager class, we are going to have "level" instance which will keep track

of the level which the user in. As the level is finished, it goes to next level.

- We have "score" instance that represents the player's score. As the game continue, the score will be updated.

- We have "time" instance in the game. When the game starts, time will upload automatically

and it starts to decrease. If the time is out and the level is not finished yet, then game is over.

- We have "bestScore" and "bestTime" instances which will be represented in statistical data

where the user can go and check those results.

- The instance "gameOn" states that game is still being played.
- The instance "gameOver" states that either time is over or the player could not place

the shapes appropriately.

- We have "mapManager" instance which belongs to MapManager class in order to show the

current layout. It will updates the game layout and let us to see the changes instantly.

- We have "inputController" instance which belongs to InputController class. This instance

will be matched with specific set of keys on the keyboard so shat game manager gets those values

and perform its task accordingly.

- startGame(): This method creates a new game and reset all the instances to default.
- finishGame(): This method ends the game if the user wants to exit from a game.
- updateScore(): This method updates a player's score as the player performs a valid move.
- pauseGame(): This method pause the game and time as well if the player pressed pause button.
- updateScore(): This method updates the player's score if the player performs a valid move.
- pauseGame(): This method stops the game and time instance until the player presses the resume button.
- continueGame(): This method resumes on the game and let the time starts to decrease.
- getSettings(): This method returns the current state of settings.
- getMenu(): This method helps to display our main menu panel.

- `startLevel()`: This method enables user to start a specific level.
- `finishLevel()`: This method ends the specific level.
- `isGameOver()`: This method returns a boolean value whether the game is over or not.
- `isGameOn()`: This method returns a boolean value whether the game is still being played.
- `isTimeUp()`: This method returns a boolean value. If time hits 0 while the game is played, it returns true otherwise false.
- `isLevelOver()`: This method checks the grid of the game, if there is no empty place to be filled, then it returns true otherwise false.

### 3.2.2 Map Manager

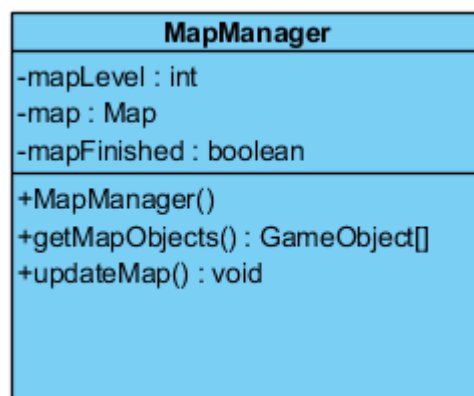


Figure 11 – MapManager class

MapManager class works with FileManager and MapManager classes. This class checks the map. It has `mapLevel` integer that holds the level, `map` is the Map object for the current level and `mapFinished` is the Boolean to check map is finished or not. It has a constructor that create MapManager object and `getMapObjects()` returns the array that hold objects in the map. Also, there is an `updateMap()` method that updates the map on each move.

### 3.2.3 File Manager

FileManager
-scannedImages : Image[] -scannedTexts : string[] -scannedAudio : AudioFormat[]
+FileManager() +getMapLevelData() +getHowToPlay() : string[] +getCredits() : string[] +getStatistics() : string[] +getSelection() : string[] +saveSetting() : void +saveSelection() : void

Figure 12 – File Manager class

FileManager class works with MapManager and GameManager. This class checks the file and retrieves the required data from the file. It has scannedImages as image array that holds images of the file. scannedTexts as string array that holds texts of the file. scannedAudio as AudioFormat array that holds audios of the game. getMapLevelData() retrieves the map data from the file which written by MapManager. getHowToPlay(), getCredits(), getStatistics(), getSelection() methods return current strings from the file.

**saveSetting() and saveSelection() methods save the current setting and level selection to the file. So, user can reach the saved settings and levels.**

### 3.3 Game Objects Subsystem

Game Screen Elements Subsystem “Game Screen Elements Subsystem” declares the objects to show in the screen while the game is running. It contains objects of “Map”, “GameObject”, “Grid”, “Shape” and “Box”.

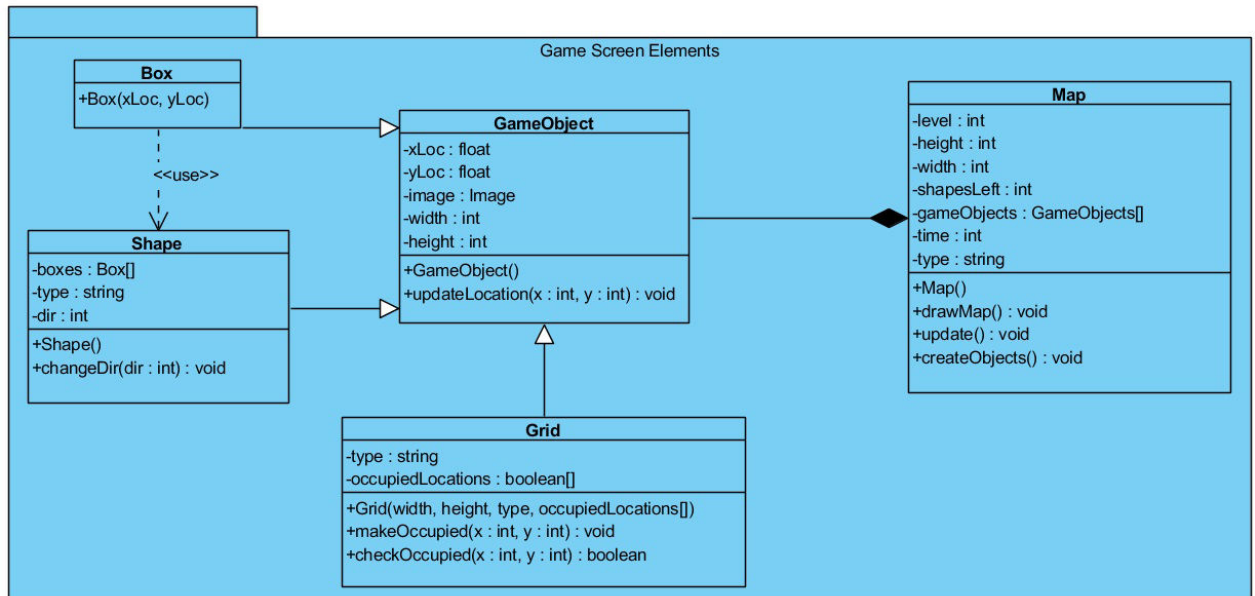


Figure 13: Game Objects Subsystem

#### 3.3.1 Map Class

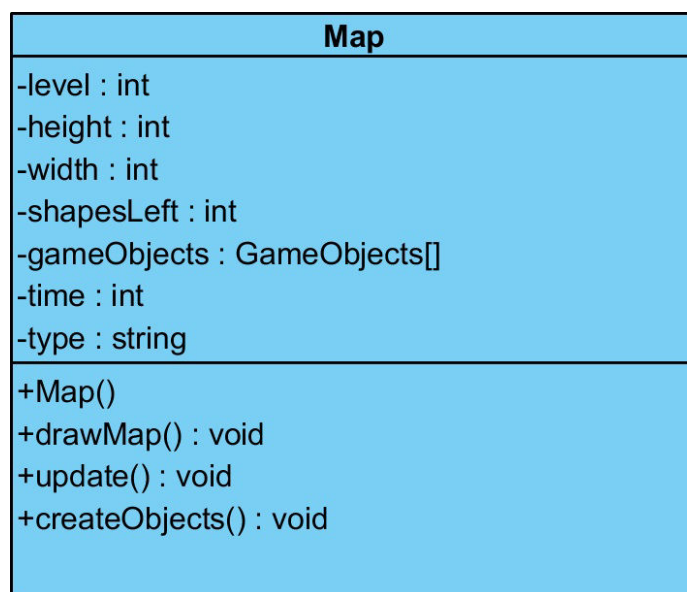


Figure 14: Map Class

“Map” class is the class that contains all GameObjects. It has attributes such as “level”, “height” and “width”. “height” and “weight” are the size informations of the map and “level” is the number of the level. “shapesLeft” is the number of shapes that are not place on the grid yet. “gameObjects” are the array of GameObjects that will be instantiated on the level. “time” is the timer value that will be started after the game is started. “type” is the type of map that can be changed according to game mode. “drawMap()” is the method that draws the map when the game is started. “update()” method updates the the map after every action according to the changes on the map that are made. “createObject()” method creates all GameObjects such as Shapes, Boxes and Grid when the game is started.

### 3.3.2 GameObject Class

GameObject
-xLoc : float -yLoc : float -image : Image -width : int -height : int
+GameObject() +updateLocation(x : int, y : int) : void

Figure 14: GameObject Class

“GameObject” will be instantiated after user starts to play the game. All fundamental objects in the game such as “Grid”, “Shape” and ”Box” use the “GameObject” abstract class as a parent class since they all need to have images, location information and size information. Variables “xLoc” and “yLoc” contains the locations information of a



GameObject. “width” and “height” contains the size information of a GameObject. “image” is the image that will be showed to the user. Since the objects can be moved during the game All GameObjects have the method “updateLocation()” takes the new location information as parameter and updates the location of a GameObject.

### 3.3.3 Box Class

Box
+Box(xLoc, yLoc)

Figure 15: Box Class

“Box” class represents the units that form the shapes. It doesn’t have any additional attributes. It has a constructor that takes the location information as a parameter and it constructs the Box object according to this information.

### 3.3.4 Grid Class

Grid
-type : string -occupiedLocations : boolean[]
+Grid(width, height, type, occupiedLocations[]) +makeOccupied(x : int, y : int) : void +checkOccupied(x : int, y : int) : boolean

Figure 15: Grid Class

“Grid” class is the field that the Shapes will be placed on. “Grid” has two additional attributes in addition to GameObject’s attributes. “type” is the is the string that represents the type of the grid. “occupiedLocations” is contains a truth value for every space on the grid. The truth value for all locations on the Grid will be false when the game starts and

when a Shape object is placed on the grid the truth values of the corresponding indices will be true. “makeOccupied()” method will perform the change of truth values.  
“checkOccupied()” method checks if the chosen locations are occupied or not.

### 3.3.5 Shape Class

Shape
-boxes : Box[] -type : string -dir : int
+Shape() +changeDir(dir : int) : void

Figure 16: Shape Class

“Shape” class represents the shapes that the player will try to place on the grid. It has three additional attributes in addition to GameObject classes attributes. “boxes” are the collection of Box objects that a Shape object is made of. “type” shows the type of the object and “dir” shows which state of a Shape object after it is rotated. “changeDirection()” method changes the position of the Shape object by rotating it.

## **4. Low-level Design**

### ***4.1 Object Design Trade-Offs***

#### ***Understandability vs Functionality:***

Our system is designed to be easy to learn and understand the game. To achieve this goal we removed some of complex functions and game modes such as 3D mode in order to eliminating confusing and complicated options. This way people would not try to understand the game rather than just simply enjoy it.

#### ***Memory vs Maintainability***

During the analysis and design phases, we tried to maintain our system with as less memory allocation as possible. We have some unnecessary methods and attributes that created for increase maintainability but while we are increasing it, these kind of features of design also decreased the efficiency in memory usage. But we tried to come up with the most efficient way as we can.

## 4.1 Final Object Design

Visual Paradigm Standard (seto@Bilkent Univ.)

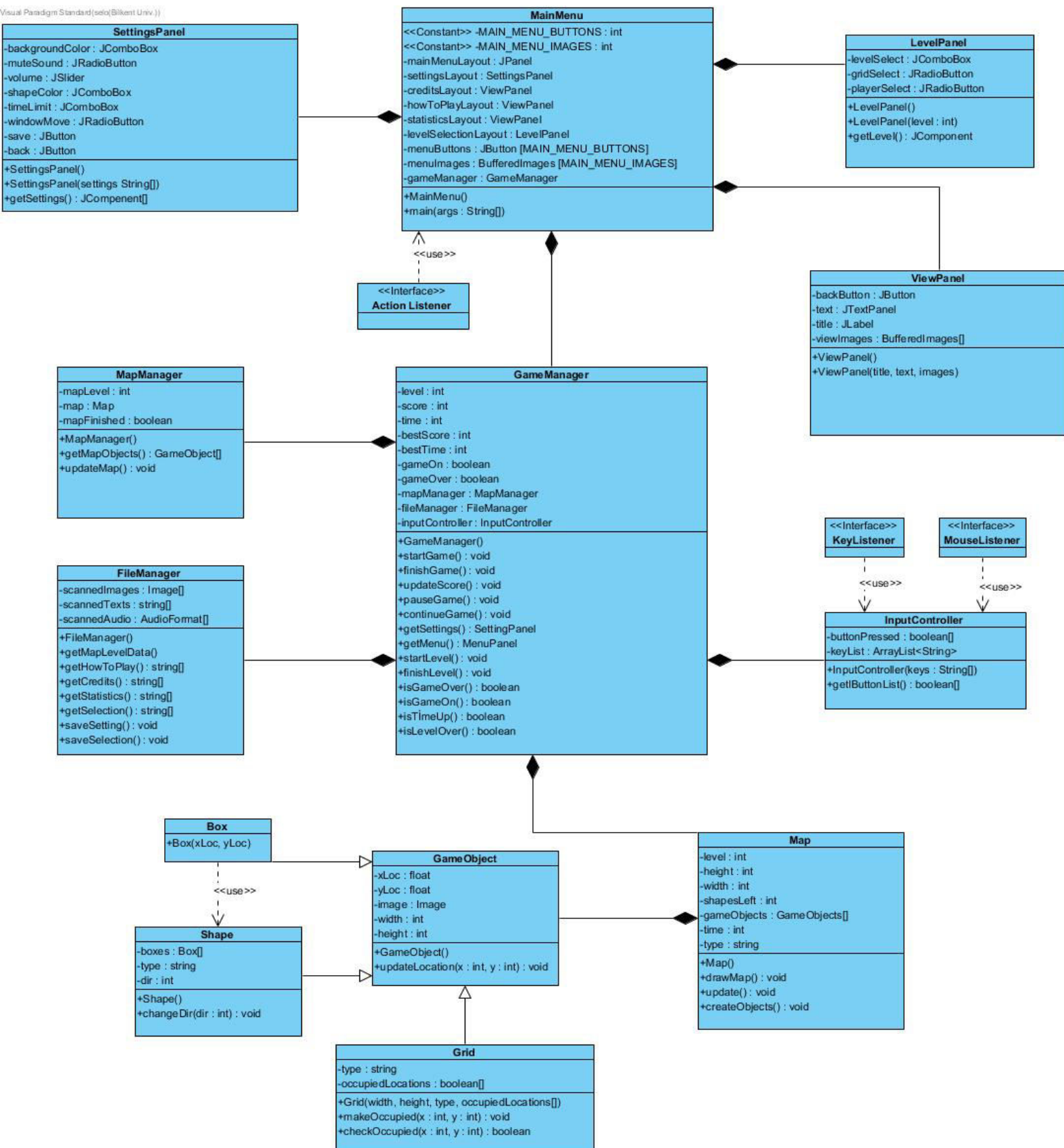


Figure 17- Final Object Design

## **4.3 Packages [1]**

### **4.3.1 *java.util***

This package contain ArrayList which will be used as mentioned at classes to control multiple variables, such as control keys of a user. IO, subpackage will be used to interact with folders and .txt files, in order to import default settings or grids.

### **4.3.2. *javafx.scene.layout***

This package provides User Interface layouts to fit GUI objects.

### **4.3.3. *javafx.scene.paint***

This package provides a set of classes to handle visual output.

### **4.3.4. *javafx.scene.events***

This package provides the controll of the events. Delivery of inputs between different hardware components and exception handling is done by this package.

### **4.3.5. *javafx.scene.input***

This package provides input handlers for keyboard and mouse,

### **4.3.6. *javafx.scene.image***

This package provides the usage of images in program. We have different kind of shapes with different colors and we will use this package to creating these shapes' image.

### **4.3.7 *javafx.scene.shape***

This package provides the set of 2D classes for defining and performing operations on objects related to two-dimensional geometry.

### **4.3.8. *javafx.geometry***

This package provides the set of 2D classes for defining and performing operations on objects related to two-dimensional geometry.

## ***4.4 Class Interfaces***

### ***4.4.1 MouseListener***

This interface will be invoked whenever a mouse action is received from user when they try to move the shapes or click the buttons on the Menu. We are implementing this interface on the InputController class which controls every action from the user.

### ***4.4.2 KeyListener***

This interface will be invoked whenever user uses the keyboard. We have 4 keyboard commands which are shortcuts of different panels. We are implementing this interface on the InputController class also.

### ***4.4.3 ActionListener***

This interface will be invoked whenever an any kind of action happens. We are implementing this interface on the MainMenu class for reducing the misses for any action that user wants to do in MainMenu and other panels.

## ***5. References***

- [1]JAVAFX Docs  
JavaFX 8, docs.oracle.com/javase/8/javafx/api/toc.htm.