IQ Puzzler

Design Report

Game Of Objects

Zafer Çınar

Mehmet Sanisoğlu

Arda Türkoğlu

Engin Deniz Kopan

Mehmet Selim Özcan

Dec12, 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.

1.1Purpose 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 o 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

JavaSwing: a software platform for creating and delivering desktop applications

JDK8: Java Development Kit 7

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 parts: Visualization, Game Engine and Data Management. Each category can easily be modified within themselves and so any additions to the project will be fairly easy to implement and keep track of. In this sense, we aimed for a model with high coherence that eliminates any unnecessary auxiliary functions and classes while providing a relatively loosely couplings between the classes. To achieve this, we preferred to use an MVC pattern to use in our system.

The MVC structure divides the implementation into three distinct parts, which will all have their own subsystems, and be easily modifiable within themselves, while providing the aimed standards. The subsystems within a part will have knowledge about each other, but will have no information regarding to the those that are outside their part.

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. In this sense, a different model with a controller will also be able to use the same View part to visualize an entirely different project.

Controller will be the GameManager 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 that are present within the Model and the visual representations of them and keep the game logic up and running. The logic game logic will be able to adapt to a change in the model and transfer the relative info to the View part.

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

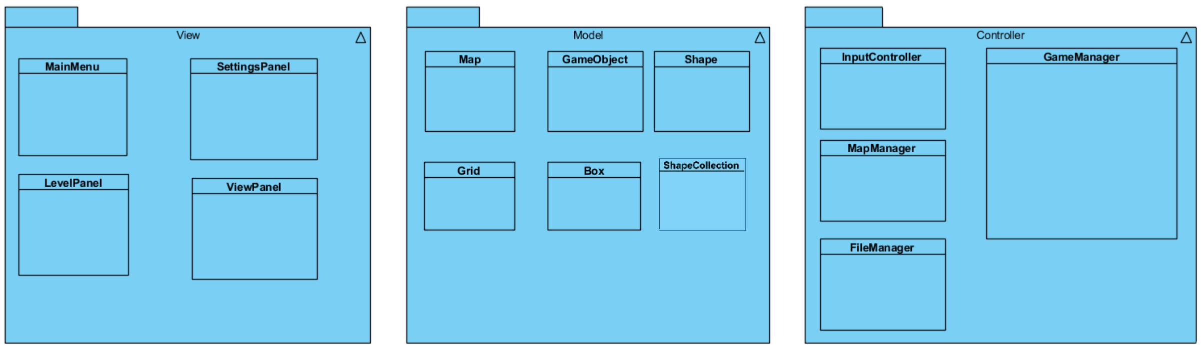


Figure 1- MVC Components of the system

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.

The subsystems will be in a hierarchical structure, that demonstrates a opaque layering model, in which each of the subsystems are only able to access the data elements of the layer that is below them. In this sense, our MainMenu class, which is placed higher up in the hierarchy will have access to the elements such as GameManager and SettingsPanel that are one level below them.

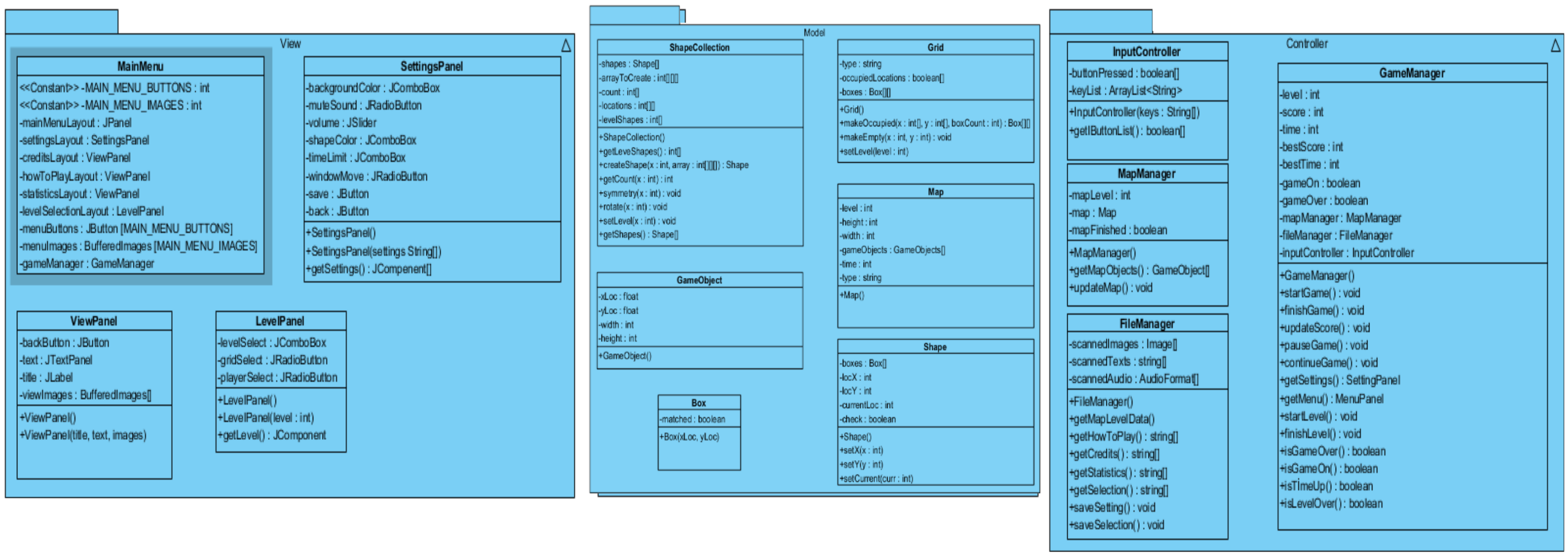


Figure 2- MVC class diagrams

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 JavaSwing, the setup will also require an updated Java Development Kit version, preferably JDK7 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 however, 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 any database storage for it to function properly. We preferred to use a filesystem since the data we manage is small and is not accessed by multiple users or from other platforms. 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 again be saved in similar .txt files and the game will present a customized playthrough to the user.

2.4 Access Control and Security

The game will treat each user as the same, as “player”, since there are no admin-specific features in the game. The game will be operating according to the currently selected settings and present a game accordingly. There are no sensitive data or user account present, so we decided not to implement any security measures to the game.

2.5 Boundary Conditions

Our implementation will be simplified in three stages of the execution of our program. At the initialization phase game presents the visual representation of the menu. From this point on, the game is ready to accept input from the player. According to the fed input, the program navigates and creates appropriate responses.

On the termination phase, the game dissolves the currently active panels first and then lastly ends the game logic itself, forcing a quit on the system. At any stage of the game, if “back” is pressed, only the most recently active subsystem will be terminated and game logic will still be up and running.

On the failure phase, our implementation does not demonstrate any fatal error conditions that could terminate the game logic unexpectedly. However, during an unexpected bug, the game is expected to restore itself by re-initializing the game logic element, “GameManager” with the data regarding the user progress, which is located in a txt.file inside the game folders.

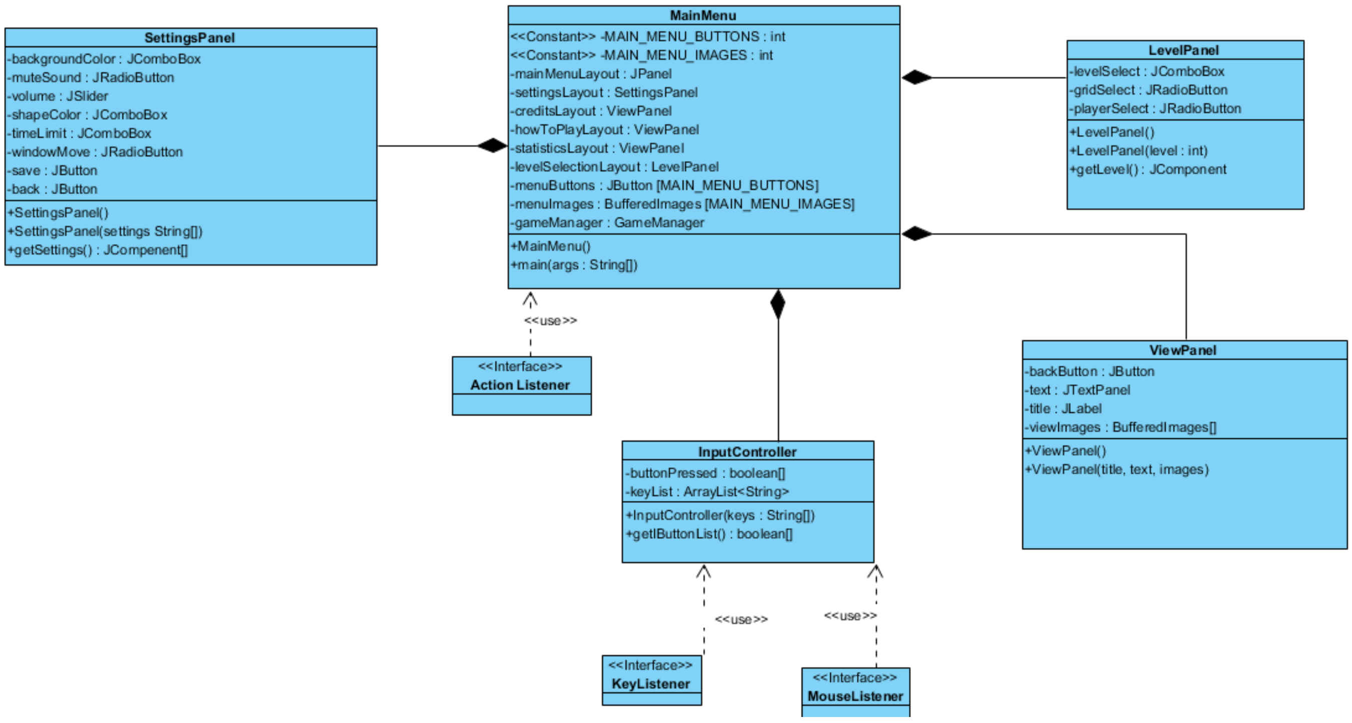
2.6 Control Flow

IQ Puzzler will demonstrate a control flow that is mainly based on events. In this event-driven flow our system depends on the user interaction to proceed with the next step in the flow of the program. According to the user interaction, an appropriate response will be created according to the listeners that we will implement, and will be passed to the View part of the game to continue the flow. The flow is decentralized in the sense that each object knows a small part of the interactions and the responsibilities of the listeners will be distributed among the manager classes such as GameManager, SettingsManager, FileManager.

3.Subsystem Services

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

3.1 User Interface Subsystem



SettingsPanel
-backgroundColor : JComboBox
-muteSound : JRadioButton
-volume : JSlider
-shapeColor : JComboBox
-timeLimit : JComboBox
-windowMove : JRadioButton
-save : JButton
-back : JButton
+SettingsPanel()
+SettingsPanel(settings String[])
+getSettings() : JCompenent[]
MainMenu
<<Constant>> -MAIN_MENU_BUTTONS : int
<<Constant>> -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[])
LevelPanel
-levelSelect : JComboBox
-gridSelect : JRadioButton
-playerSelect : JRadioButton
+LevelPanel()
+LevelPanel(level : int)
+getLevel() : JComponent
<<use>>
<<Interface>>
Action Listener
ViewPanel
-backButton : JButton
-text : JTextPanel
-title : JLabel
-viewImages : BufferedImages[]
+ViewPanel()
+ViewPanel(title, text, images)
InputController
-buttonPressed : boolean[]
-keyList : ArrayList<String>
+InputController(keys : String[])
+getIButtonList() : boolean[]
<<use>>
<<use>>
<<Interface>>
KeyListener
<<Interface>>
MouseListener

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

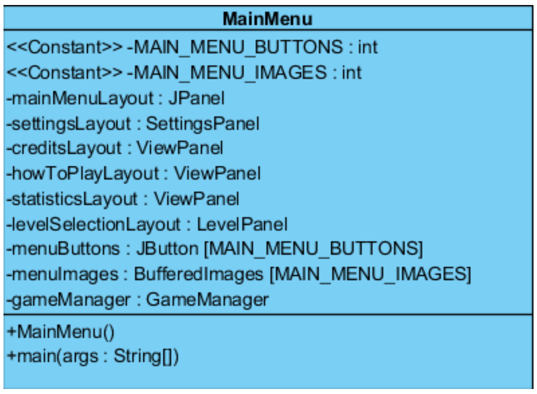

MainMenu
<<Constant>> -MAIN_MENU_BUTTONS : int
<<Constant>> -MAIN_MENU_IMAGES : int
-mainMenuLayout : JPanel
-settingsLayout : SettingsPanel
-creditsLayout : ViewPanel
-howToPlayLayout : ViewPanel
-statisticsLayout : ViewPanel
-levelSelectionLayout : LevelPanel
-menuButtons : JButton [MAIN_MENU_BUTTONS]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

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

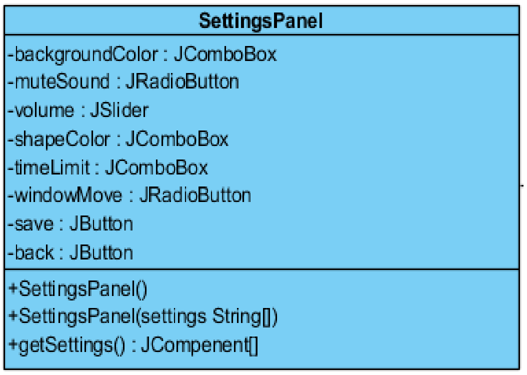

SettingsPanel
-backgroundColor : JComboBox
-muteSound : JRadioButton
-volume : JSlider
-shapeColor : JComboBox
-timeLimit : JComboBox
-windowMove : JRadioButton
-save : JButton
-back : JButton
+SettingsPanel()
+SettingsPanel(settings String[])
+getSettings() : JCompenent[]

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

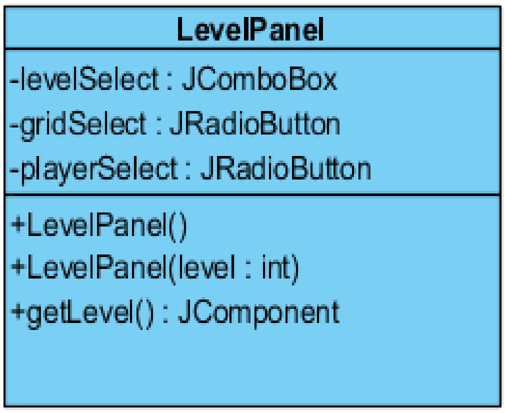


Figure 6 – Level Panel

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

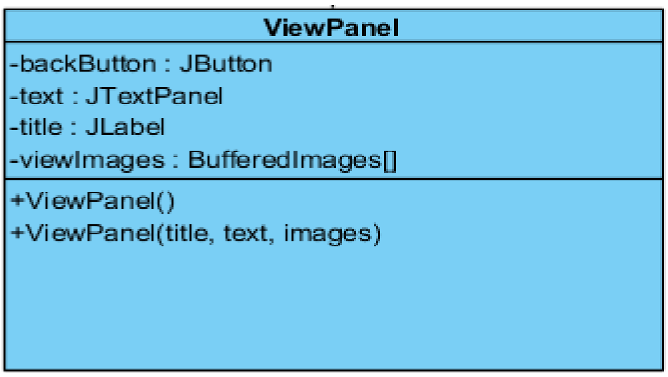


Figure 7 – View Panel class

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

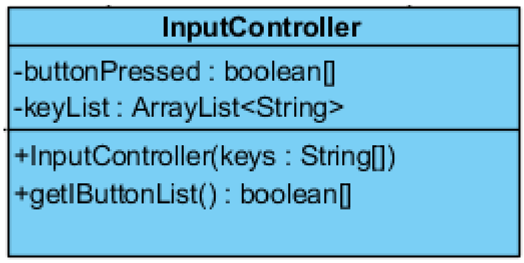

InputController
-buttonPressed : boolean[]
-keyList : ArrayList<String>
+InputController(keys : String[])
+getIButtonList() : boolean[]

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 buttonsPressed array to the user.

3.2 GameManager Subsystem

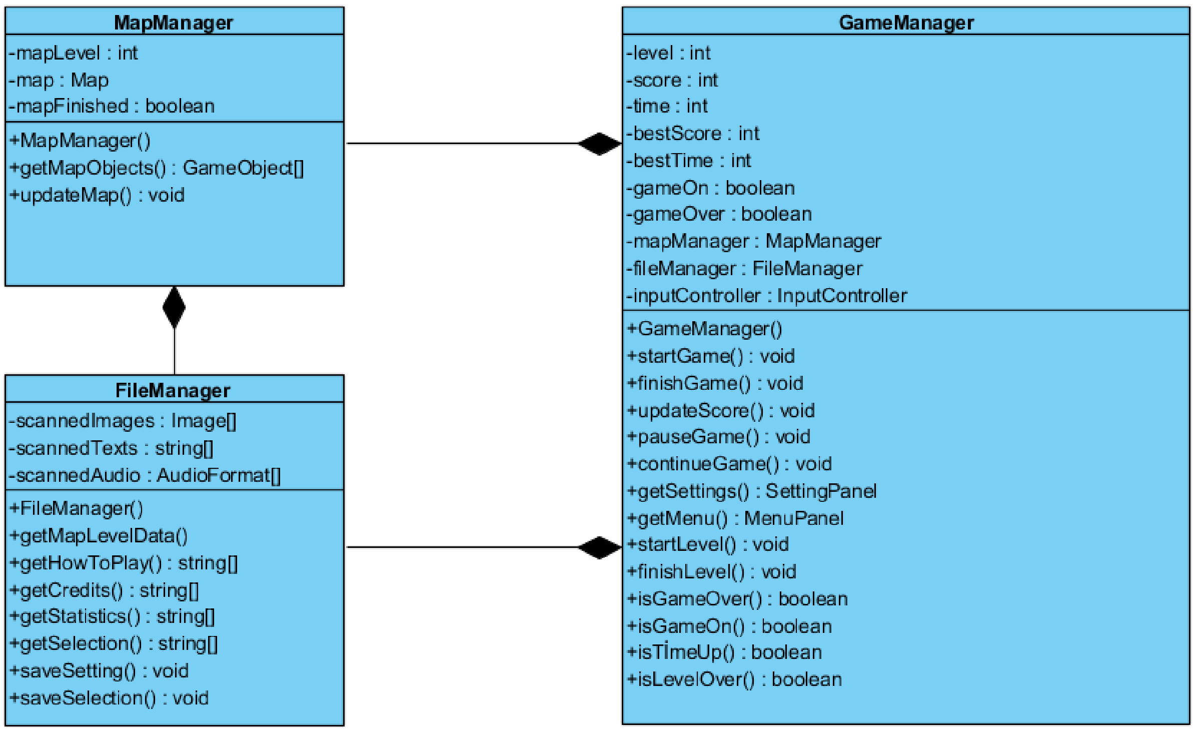


Figure 9 – Game Manager Subsystem

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

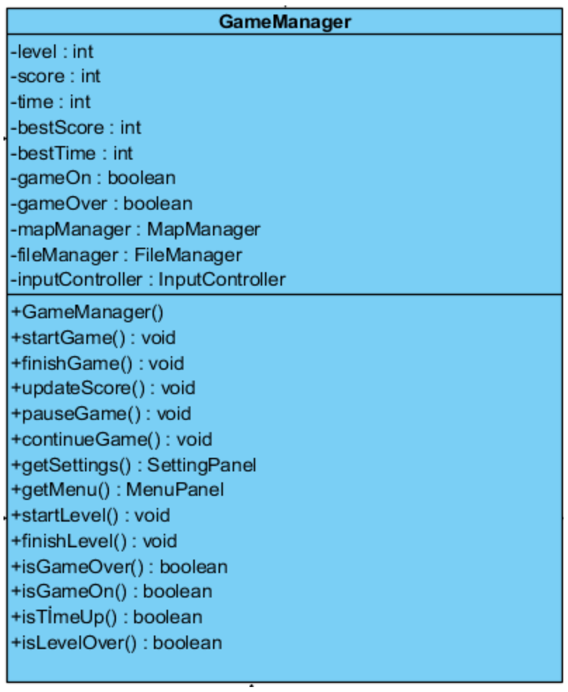


Figure 10 – Game Manager class

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 keybord 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

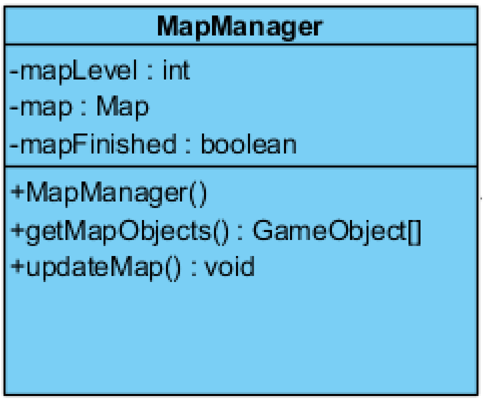


Figure 11 – MapManager class

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

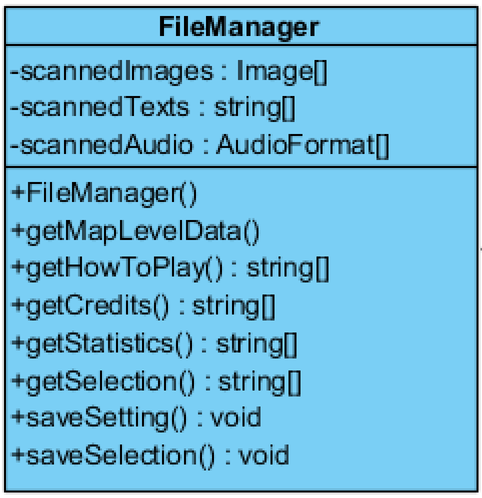


Figure 12 – File Manager class

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 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", "ShapeCollection" and "Box".

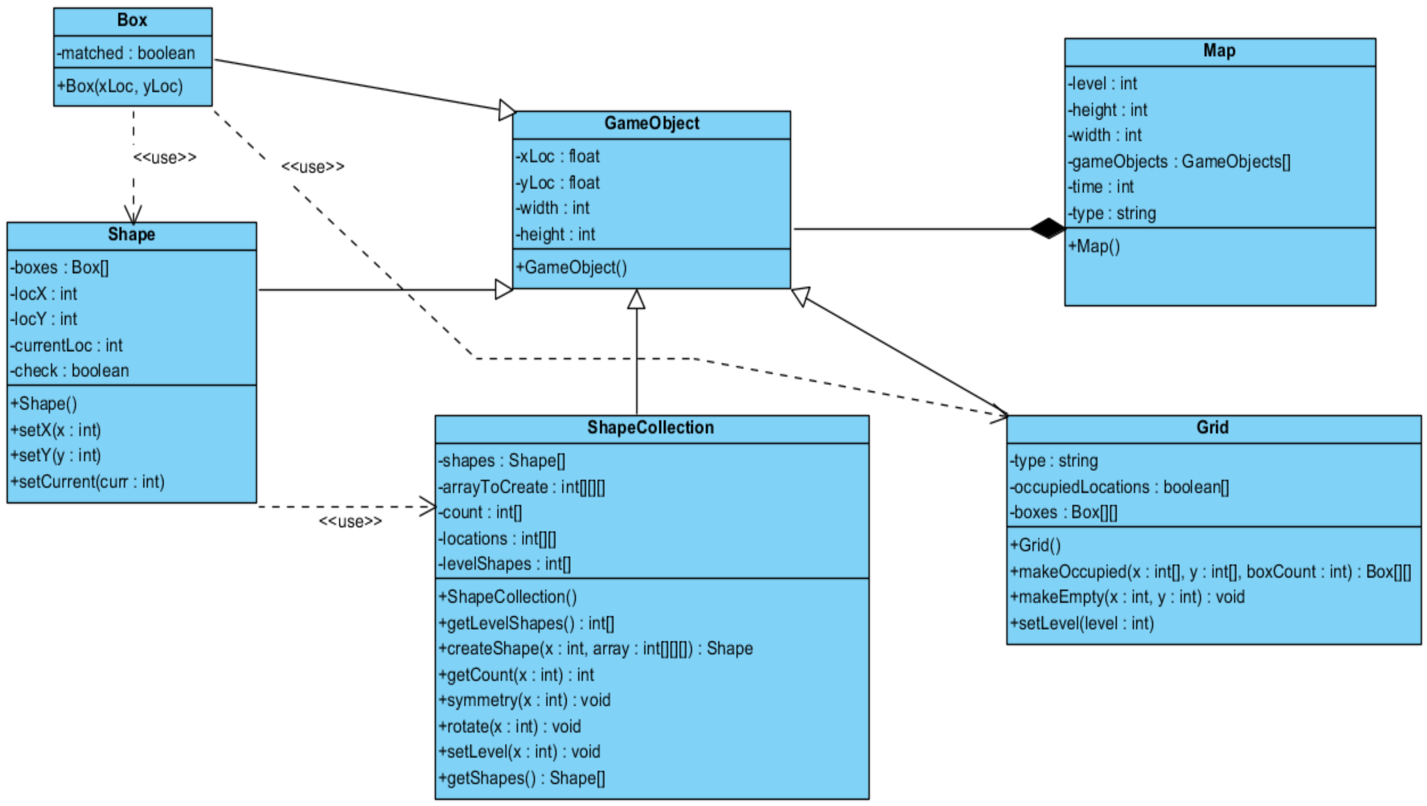


Figure 13: Game Screen Elements Subsystem

3.3.1 Map Class

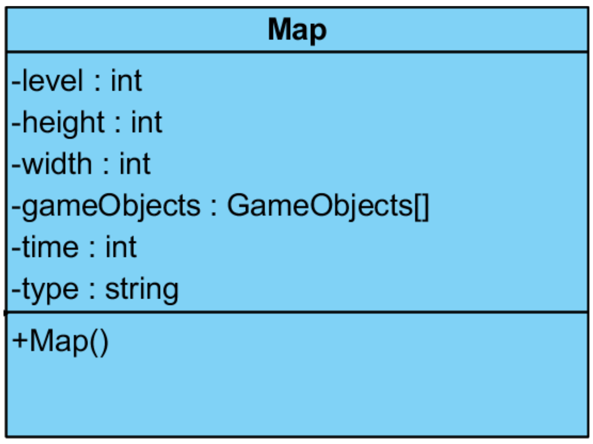


Figure 14: Map Class

“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. “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.

3.3.2 GameObject Class

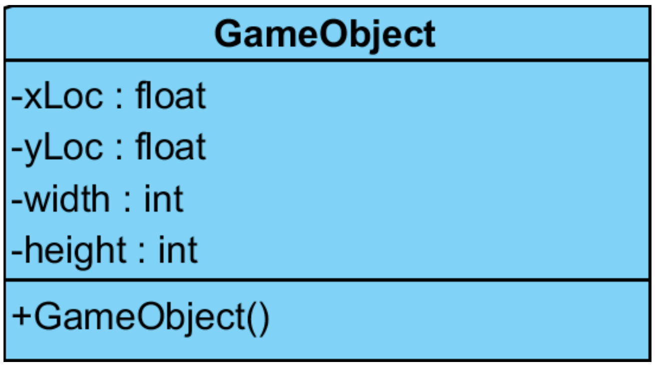


Figure 15: GameObject Class

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

3.3.3 Box Class

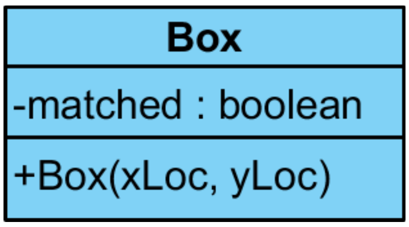


Figure 16: Box Class

“Box” class represents the units that form the shapes. It has a boolean variable named matched. “matched” represents whether a box on a grid is occupied or not. “matched” variables hold the truth value “false” for the “Box” objects that represent the empty spaces on a grid. But for the occupied ones and for the shapes, the value is “true”. 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

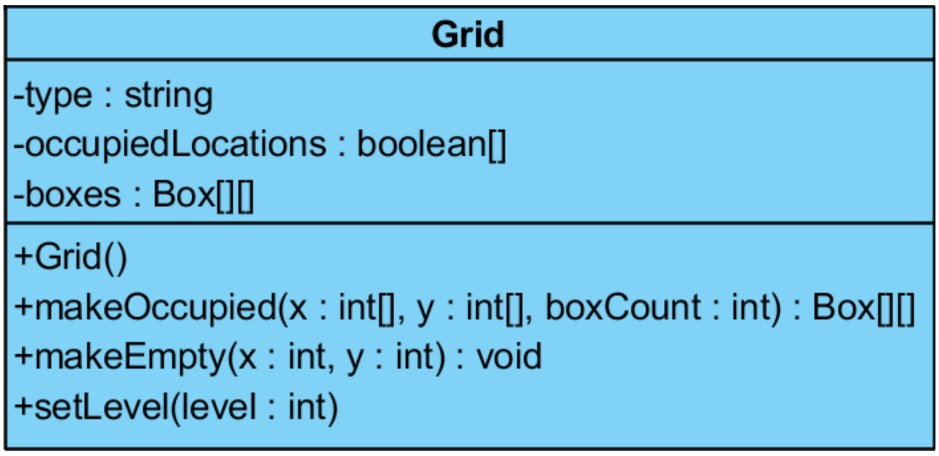

Grid
-type : string
-occupiedLocations : boolean[]
-boxes : Box[][]
+Grid()
+makeOccupied(x : int[], y : int[], boxCount : int) : Box[][]
+makeEmpty(x : int, y : int) : void
+setLevel(level : int)

Figure 17: Grid Class

“Grid” class is the field that the Shapes will be placed on. “Grid” has three 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. It also has an array named “boxes” and this array contains all of the “Box” objects that represent the spaces on the grid. “makeOccupied()” and “makeEmpty()” methods perform the change of truth values of the “matched” attribute of the “Box” objects. “makeOccupied()” changes the truth value of the corresponding “Box” objects to true. “makeEmpty()”changes the truth value of the corresponding “Box” objects to false.

"setLevel()" method takes the number of the current level and initializes the grid by filling the corresponding boxes according to the level number.

3.3.5 Shape Class

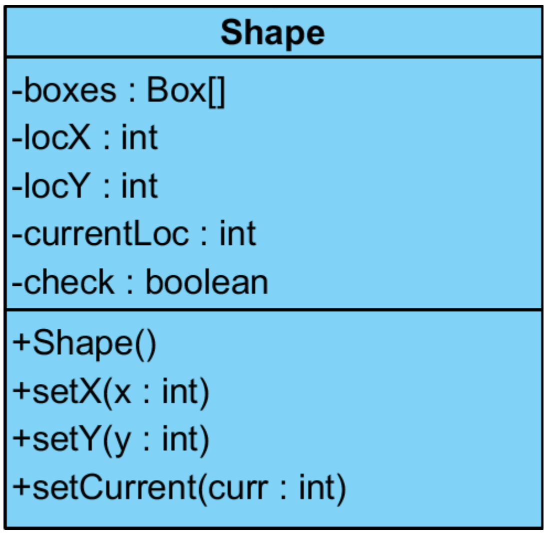


Figure 18: Shape Class

"Shape" class represents the shapes that the player will try to place on the grid. Variables "locX" and "locY" contains the location information of a Shape. "currentLoc" hold the current location of the "shape" object. "boxes" is the collection of Box objects that a Shape object is made of. "setX()" and "setY()" methods change the values of "locX" and "locY". "setCurrent()" updates the current location of the shape.

3.3.6 ShapeCollection Class

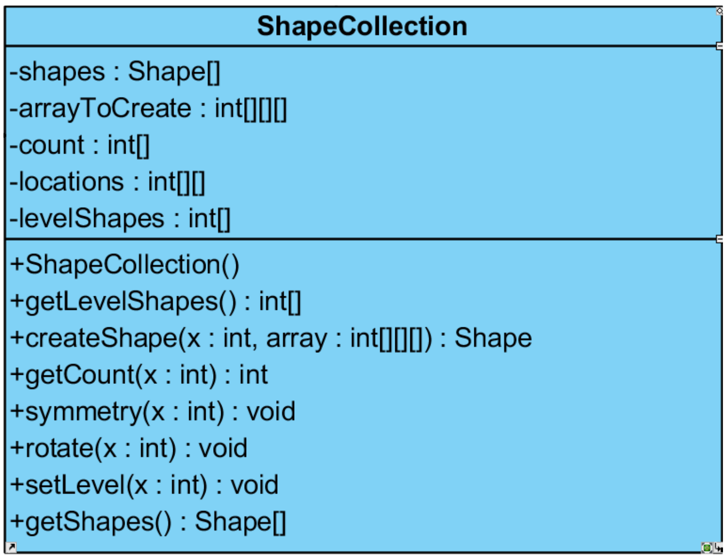


Figure 19: ShapeCollection Class

"ShapeCollection" class represents the all of shapes that the player will use on a level. "shapes" is an array of "Shape" objects. This array contains the collection of shapes that will be used on a level. "arrayToCreate" holds the information that will be used to create the shapes on a level. "count" is the number of boxes for each shape in the collection. "levelShapes" is an integer array and this array contains the information of which types of shapes will be created on a chosen level. "getLevelShapes()" returns the "levelShapes" array. "createShape()" method initializes "Shape" objects according to which shape they will be."getCount()" returns the number of shapes in the collection. "rotate()" method rotates a chosen shape 90 degrees. "symmetry()" method flips a chosen shape. "setLevel()" method takes the number of the level that will be played and initializes the shapes by creating the shapes that will be used on that level. "getShapes" method returns the shapes in the collection.

4. Low-level Design

4.1 Object Design Trade-Offs

Understandibility 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.

Efficiency vs Functionality

In our design we have some inefficient factors in order to increase functionality of our game. Although our attempt of making the game as optimum as we can, we had to make some sacrifice in efficient designing. Since functionality is more important in our game for clients rather than efficieny in our design, this trade-off was necessary.

4.1 Final Object Design

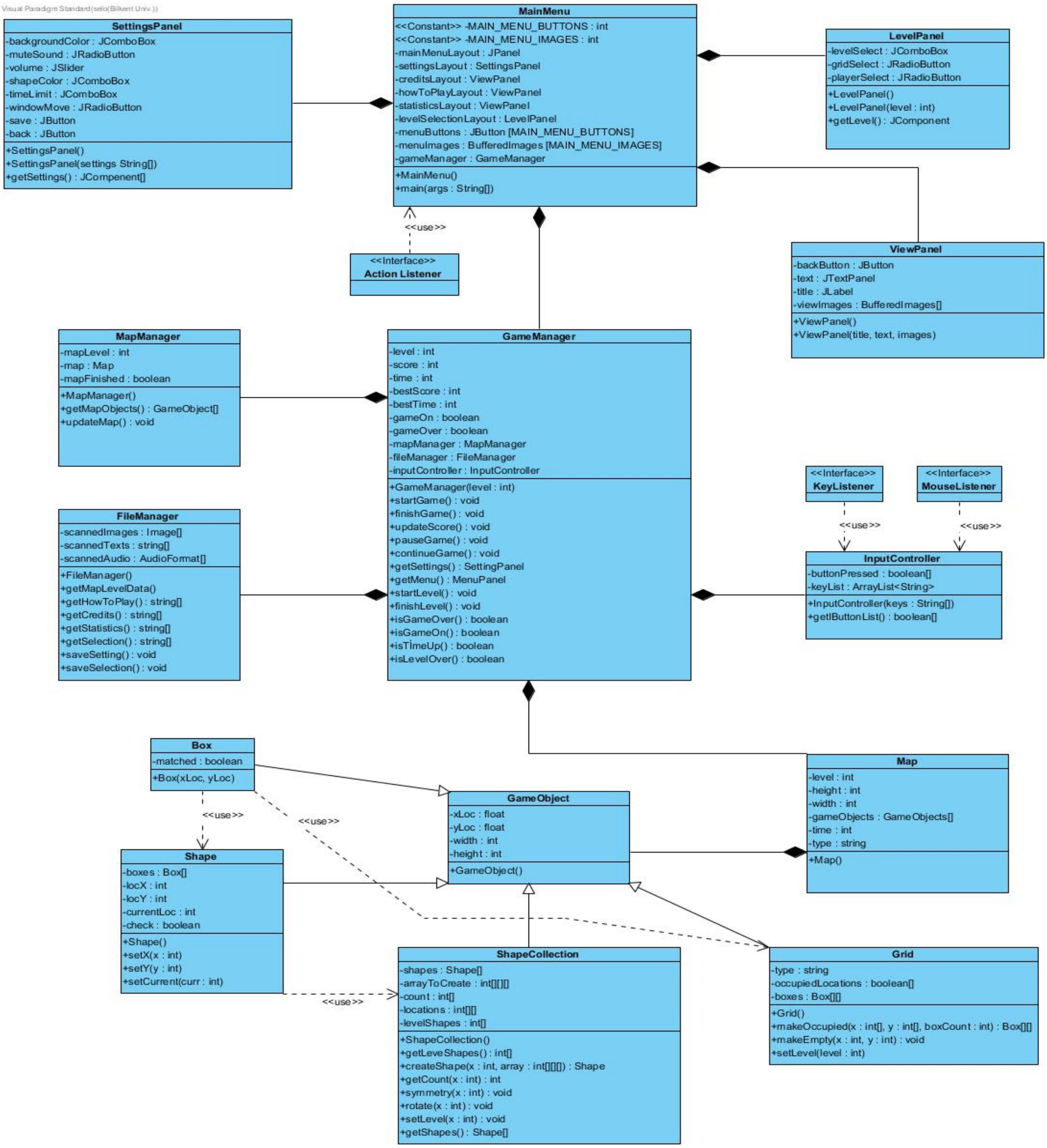


Figure 20 - Final Object Design

In Figure-17 we have our final object design that we are currently implementing. It includes all classes that are explained above.

4.3 Packages

We use packages that provided from Java Development Kit for creating our game.

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. javax.swing

This package contains the core Swing components, including most of the model interfaces and support classes.

4.3.3. java.awt

This package contains all of the classes for creating user interfaces and for painting graphics and images.

4.3.4. java.awt.Graphics2D

This package defines a device-independent interface to graphics. It specifies methods for doing line drawing, area filling, image painting, area copying, and graphics output clipping.

4.3.5. java.awt.event

Provides interfaces and classes for dealing with different types of events fired by AWT components.

4.3.6. java.awt.color

This package provides classes for color spaces.

4.4 Class Interfaces

We have 3 Interfaces for our game to be user controllable. These interfaces are already implemented in Java Documentation and they are listeners.

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. *Improvement Summary*

- We have changed our final object design by adding ShapeCollection class and changing some of already existed classes.
- We have decided final high-level software architecture
- We have finalized our boundary conditions, control flow and mainly subsystem decompisition
- We have changed our design by using Java Swing instead of using JavaFX
- We have added one more trade-off which is efficiency vs functionality.

6.References

- What Every Computer Scientist Should Know About Floating-Point Arithmetic, docs.oracle.com/javase/7/docs/api/.