Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering

CS-319: Kill The Bugs

Design Report

Anisa Llaveshi, Cüneyt Erem, Ertunç Efe, Mert Gürcan

Course Instructor: Uğur Doğrusöz

Progress / Final Design Report
April 9, 2016

Table of Contents

1.) Introduction

1.1 Purpose of the System

The purpose of this system is to provide an entertaining game for people who enjoy playing games for all ages. The game consists in a set of bugs and a number of different food products initially placed in opposing sides of a field. The bugs aim to eat the food and the player’s aim is to not let the bugs reach them. The player can buy different weapons by collecting coins and place them in the field to kill the bugs. The game ends when a bug reaches one food product. When the game end, if the score that player earn is in the top ten highest score, the score will be recorded locally in order to show in high scores page. In addition to the basic gameplay the user will be offered with additional features like changing the game theme, changing game volume, and changing the background music volume.

1.2 Design Goals

End User Criteria

Ease of Learning

The game will be significantly easy to be learned to play. There will be a help section which can be reached from the main menu which will guide the user through the basics of the gameplay and of the features that it provides. In addition, the object images will be self-explanatory and intuitive for beginner players to understand their functionality.

Ease of Use

Any person will be able to play the game regardless of previous experience in gaming. The game will not be very difficult to play. A reasonable number of features will be available for the player to use so that the game does not become very complex in terms of difficulty to remember everything and it still remains manageable for the player to play and win high scores.

User Friendliness

The game is designed to be user-friendly.The game does not require any setup, it will be provided as a .jar file which can be executable within the java runtime environment (JRE) [1]. In that way, game can be played among different operating systems.Moreover, its main menu design and the game screen is easy to understand and easy to use which are also important aspects of user friendliness criteria.

Performance Criteria

Run–Time Efficiency

Another important issue is the runtime efficiency of the game, player will not be affected negatively from the gameplay screen while s/he is playing the game. The functions that are provided to the

player in the gameplay screen will perform reasonable time, such as buy weapon buttons, placing the weapons to the field etc.

Optimized Memory Usage

Memory usage is one of important performance criteria which depends on speed also. In this project, there are lots of animation tools which are bugs, weapons, bullets, but also their explosion animations and updates all images and scores. Therefore, the game cannot have less memory usage and need to have enough space to improve performance and speed, java virtual machine (JVM) will delete destroyed objects such as bullets, bugs etc [2].

Maintenance Criteria

Readability

Readability is an important thing that engineers always should write their code in readable because when other engineers look at the codes, they should understand the flow of methods and operations in codes fast. This provides maintenance of the code, time, efficiency. In this project, this design will based on maximum readability first, after optimizing maximum performance criterias, maximum performance and readability will be combined and have maximum readability and performance.

Modifiability

Modifiability is an important criteria for mordern systems to keep cost of the system at a reasonable level and also to gain time when updating the system.In Kill the Bugs game, MVC (Model-View-Controller) pattern will be used to maintain modifiability [3]. In that way, the game's graphical objects, algorithm and specialities of the objects can be updateable separately. Moreover, in the testing phase, fixing the problems will not be take so much time, since the causes can be easily find. For instance, if the problem will occur about the control of the game, it will be searched into the control objects and corrected as soon as possible.

Good Documentation

Good documentation is one of the key features of maintenance criteria. Comments will be written in order to make the coder more familiar to project. Also methods and variable names will be self describing. And explanation will be made how code fits into the system process. So, code should be proper for the documents and its purposes. The code may be more elegant than clever to keep it simple and understandable.

2.) Software Architecture

Model-View-Controller (MVC) architecture has been chosen for the design of the game, because it is time-efficient and it fits the game's structure [4]. In figure 1, the swimming lanes show the corresponding parts of the MVC pattern. Inside the lanes it can be seen that several components have designed for the sake of understandability and ease of implementation.

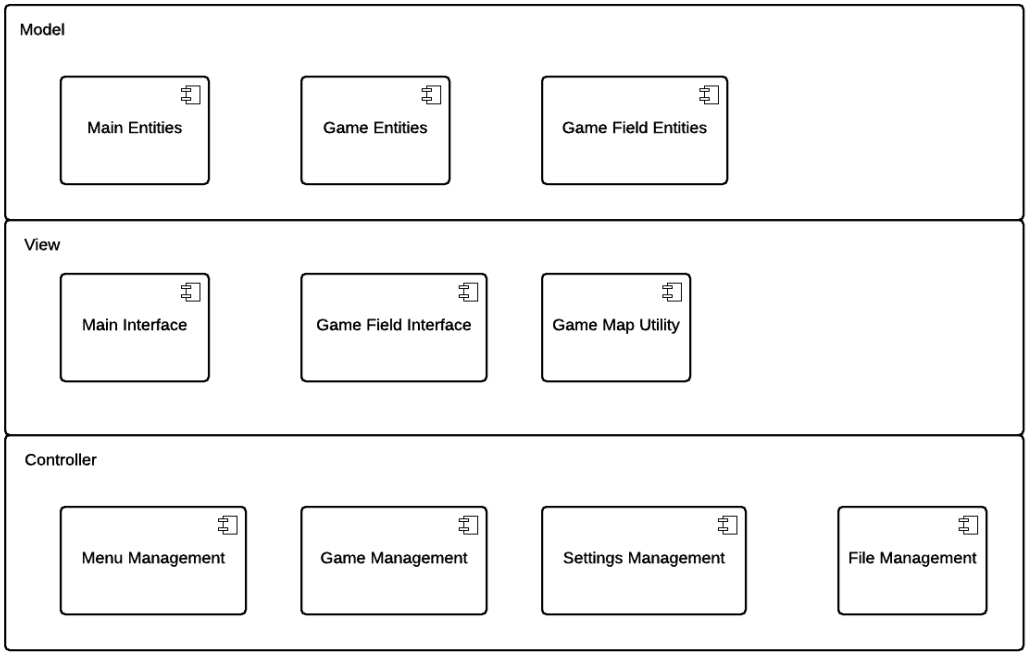

Figure 1 : Subsystem Components with Swimming Lanes for MVC pattern [5]

2.1 Subsystem Decomposition

Main Entities component is composed of the models which keep the data for the background music and similar game features needed during the whole gameplay. Such model classes are Sound, Theme, Button and User class.

Game Entities component has the game objects' models that keep game tool data which are not relevant to the actual game field but which the user needs for the gameplay. Such game tools classes are Score class, MoneyAccount, BuyButton class etc.

Game Field Entities component has the actual game field objects' models that keep objects' data which the user uses during game. Some of them are Weapon class, Bug class, Bullet class, Grenade class etc.

Main Interface component has the main view objects of the program. It displays the menu and its submenu options to the user like CreditsView, SettingsView, HelpView, etc.

Game Field Interface component has the actual game screen view objects of the program which consists of WeaponView, BugView, CoinView, GroceryView, ScoreTable, MoneyAccountTable, BulletView, and BitsNPiecesView. It displays the game play screen to the user.

Game Map Utility component has the view objects that actually need to be inherited from the Game Field Interface. It provides the methods that are needed from the game field objects to access their coordinates and to mutate their coordinates into the game field.

Settings Management component has the control classes which controls the sound and theme by using the provided services from the Model and View components.

File Management component has the control objects of the data management that provides taking data from the relevant entity object to be held into .txt file locally which are user name, high score, and game duration.

Game management component has the control classes for managing the game playing stages like, GameManager class. Also ,it has the control objects which controls the all main menu selection and also other menu selections such as, newGameButton action listener, creditsButton action listener, backButton action listener etc.

These components' services will be explained in more detail at section 3.Subsystem Services part.

2.2 Harware / Software Mapping

Our game will be a Desktop based game and a user will only need a computer in order to be able to play; no internet connection is needed. We are going to develop this software in Java Programming language thus, a user must have Java Runtime Environment (JRE) installed in his/her PC [6]. In addition, a mouse is needed in order to be able to interact with the game. A keyboard is optional as it will have shortcuts to pause the game and mute the menu.

2.3 Persistent Data Management

Kill the bugs game does not consist a high-level database structure. It consists of simple text file which is used for high score list. Text file is kept inside the kill the bugs game’s jar file. Also game sounds and game images are stored inside the jar file.

2.4 Access Control and Security

There will be no restrictions on control access of the game. Anybody who possesses the executable file of the game can have access to it. In addition we do not manage a user profile system. As a result, there will be no security provisions.

2.5 Boundary Conditions

The system will be initialized when a player will open the executable file (.jar). The system shuts down when the user will click the quit button in the game, the exit button of the window or in case the games shut down abruptly due to errors. A possible error that might occur is corruption of the data file holding the high scores of the users who have played in that machine. In such a case, the saved data will be lost and a new file holding the highest scores from that moment will be created. Another possible error is an unexpected brute shut down of the game while a player is playing. In such a case, the score of the current player will not be saved or evaluated to be put in the high score table.

3.) Subsystem Services

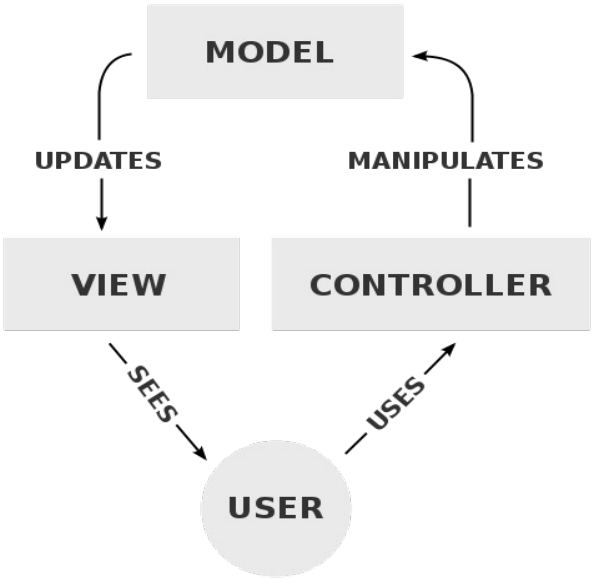

Figure 2 : MVC Pattern Usage [7]

This project will follow the MVC pattern [8].

The reason why we chose MVC pattern is because we wanted to separate the logic of the game from the views of the game. In this way we have a clean way of representing all the components of our game and separate the modelling of our game objects from the representation.

There will be three control components which make possible the settings, file and game management which are Settings Management, File management and game management components.

The view components are the Game Field Interface, Main Interface and the Game Map Utility. The Game Field Interface will provide the graphical interface for all the gaming tools and Main Interface for the main menu. Game map utility component is also a view component but this component will be inherited by the game field interface subsystem.

Model system keeps the data of the objects which are main, game and gamefield entities to calculate and update logic of the game. Model, view and controller systems are connected to each other separately to regulate the system but these connections will not affect MVC pattern negatively, all connections have been made appropriately according to the MVC pattern which can be seen in figure 2.

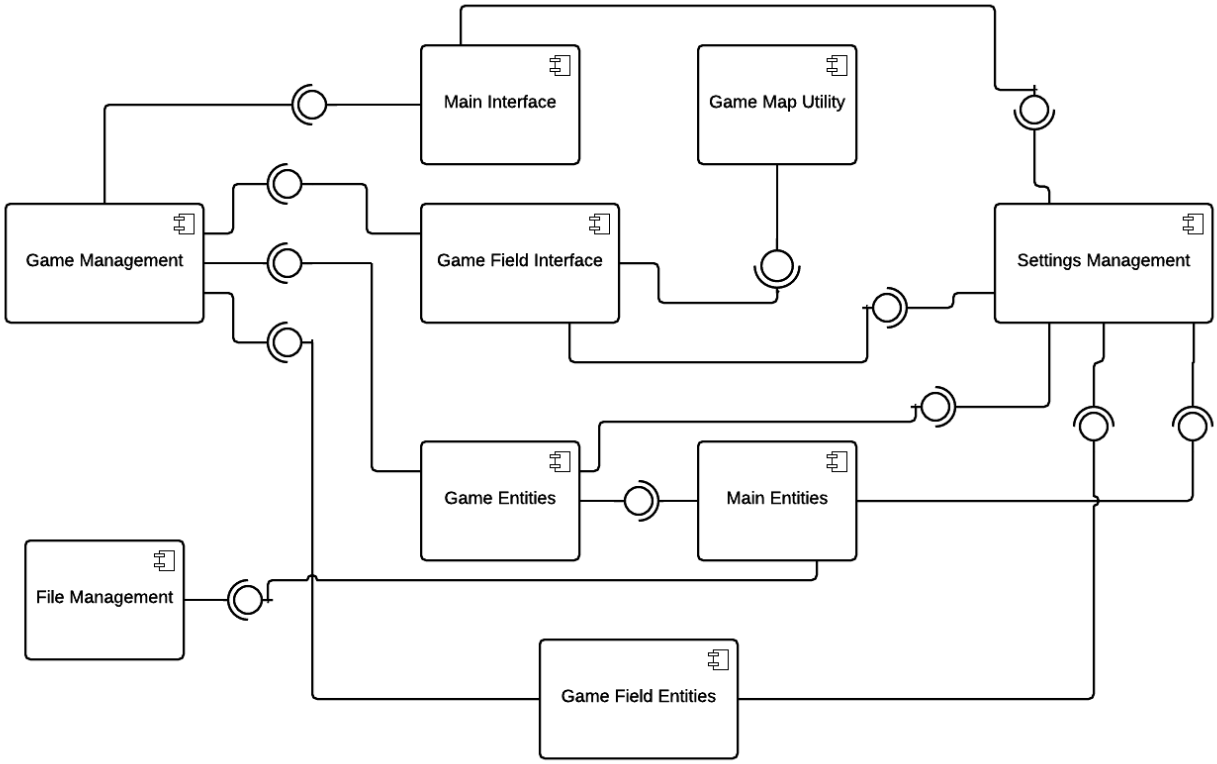

Figure 3: Component Diagram of the Project [9]

Settings Management subsystem takes input from Main Interface and Game Field Interface, which are subsystems of the view component and holds the data to the relevant Entity objects by using the provided services. Moreover, this subsystem has the ability to change the sound of objects, to adjust the selected theme by taking the relevant data from Main Entities, Game Field Entities, Game Entities subsystems and update the Interface subsystems by using their services.

File Management subsystem interacts Main Entity subsystem which have game entities' data to getting username, score and duration by using Main Entity subsystem services.

Game Management subsystem interact with Game field entities by manipulating the data of objects of game like, bugs, weapons, etc. Also, it keeps and updates data of Game Entities subsystem which are Score class and Money class by using the provided services from these subsystems. Moreover, it controls the display of Game Field Interface by using the services of Game Field Interface, some of Game Field Interface services's base operations will be taken from Game Map Utility subsystem's services via inheritance. Also, this subsystem controls the Main interface by using the services that are provided from Main Interface subsystem like, changing menus using buttons.

In the explanation part of services, management components' sockets (figure 3.2) and the provided services by the other components will be explained under the headers of services separately to maintain the simplicity.

Services

Change Background Sound is a service provided by the Game Entities component through methods to set and/or update the background music volume of the game.

Change Object Sound is a service provided by the GameField Entities component through methods to set and/or update the music and the music volume of different game objects.

Change Theme is a service provided by the Main Entities Component through methods to set and/or update the theme of the game. It is the Settings Management component which uses these services to modify the data of the entities accordingly and makes possible the update of the respective interfaces.

Change Highest Scores service is provided by the Main Entities component through methods to set and/or update the score of a user, to check whether a user's score is among the top 10 highest scores and to update the list of the users with the highest scores. The File Management component uses this service to update the files keeping the high score accordingly.

Change Score service is provided by the Game Entities component with methods to set/update the score. These methods are then used by the Game Management component which updates the score accordingly and also by the Main Entities which updates the score of a certain user.

Update Money Account service is provided similarly as the Change Score service by the Game Entities component. This service is then used by the Game Management Component.

Buy gun service is a service provided by the Game Management component by a method which taking as an input the gun information, checks if the money account has enough available money to be able to buy the gun. In addition, other methods to provide this service is a method to make the certain gun required available and another to update the money account accordingly after the gun has been bought. Game Management then also, updates the view component of the game field accordingly.

Collect coin is a service provided by the Game Management component through methods to update the money account according to the value of the coin taken as an input, and to update the game field view to make the coin disappear.

Manage Collisions is a service provided by Game Management component through methods to manage the collisions. This method uses the check collision service which is provided by Game Management component and makes the necessary decisions. It checks among all bullets and bugs in the game and if a bullet is colliding with a bug, it calls a method to decrease bug's health and checks whether the bug is alive after that or not.

Check Collision is a service provided by Game Management component through methods to check if two objects have collided with each other. This method takes as an input the location of two objects and decides whether their positions overlap.

Pause/Continue Game service is provided by Game Management component through methods to freeze and unfreeze the current status of the game. Also Game Management updates the game field interface accordingly.

End Game is a service provided by Game Management component through a method to check if any of the conditions which ends the game are fulfilled and methods to load a new game.

4.) Low-Level Design

4.1 File Management Package

FileManager is a controller class which will use the singleton pattern. Only one instance of FileManager will be possible to be created. The class will be responsible for manipulating and recording locally the data of the highest scores of all the users that have played in one machine.

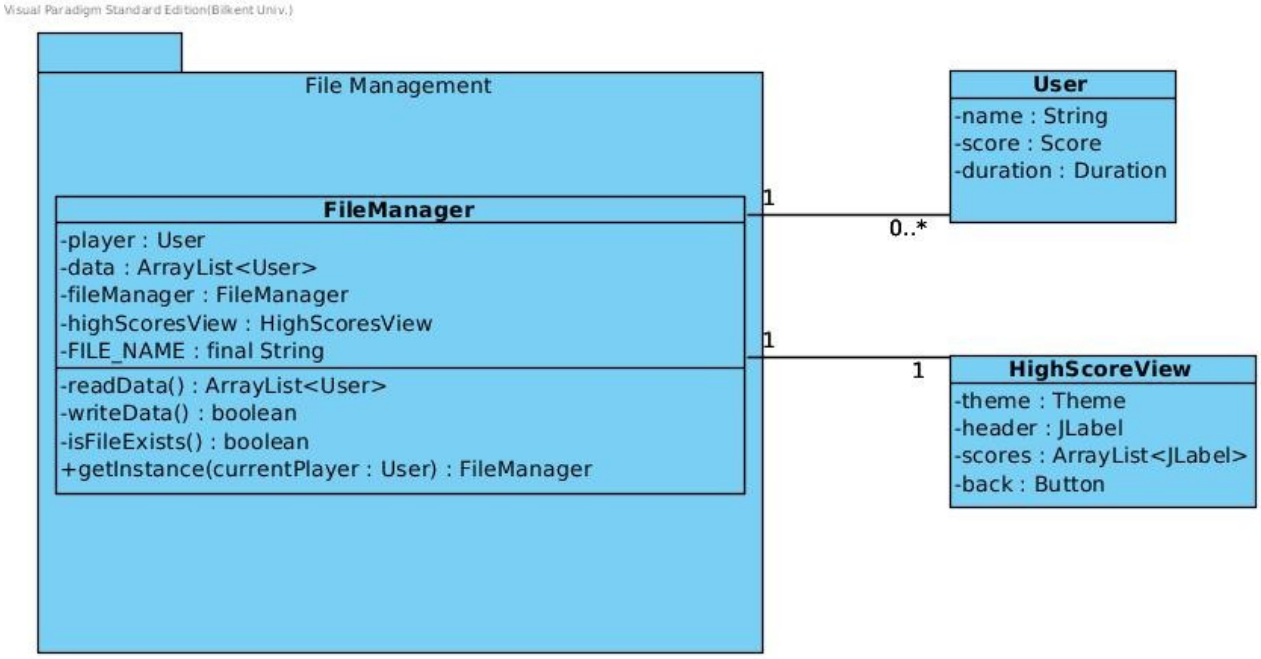

Figure 4: UML diagram of FileManager class

The attributes that this class is going to have are:

- fileManager : FileManager - an instance of itself to enforce the singleton pattern

- player : User - the current user playing the game.

- data : ArrayList<User> - all the users that have previously played the game in that machine.

- FILE_NAME : final String - a final string type to hold the name of the file that is going to hold the data.

- highScoresView : HighScoresView - an instance of the highscores view which will be updated accordingly when a game is over.

- FileManager() The constructor will be private in order to be able to implement the singleton pattern.

The methods that it is going ot have are:

- isFileExists():boolean - determines whether a text file holding previous data has already been created

- readData():ArrayList<User> - it returns the list of users which can be found into the text file

- writeData():boolean - it updates the text file with the data of the current user if the current user reaches the high score

+getInstance(User) :FileManager - method which enforces the singleton pattern. In order for a new instance of a FileManager to be created this instance must be called. This returns a new instance of FileManager in case it has not been created before and in case it has it returns the current FileManager instance.

4.2 Settings Management Package

SettingsManager is a controller class which similarly to the FileManager class will use the singleton pattern to enforce the initialization of only one instance of SettingsManager. This class will implement the ActionListener interface. SettingsManager is responsible to change the theme view of the game, and the sound volumes.

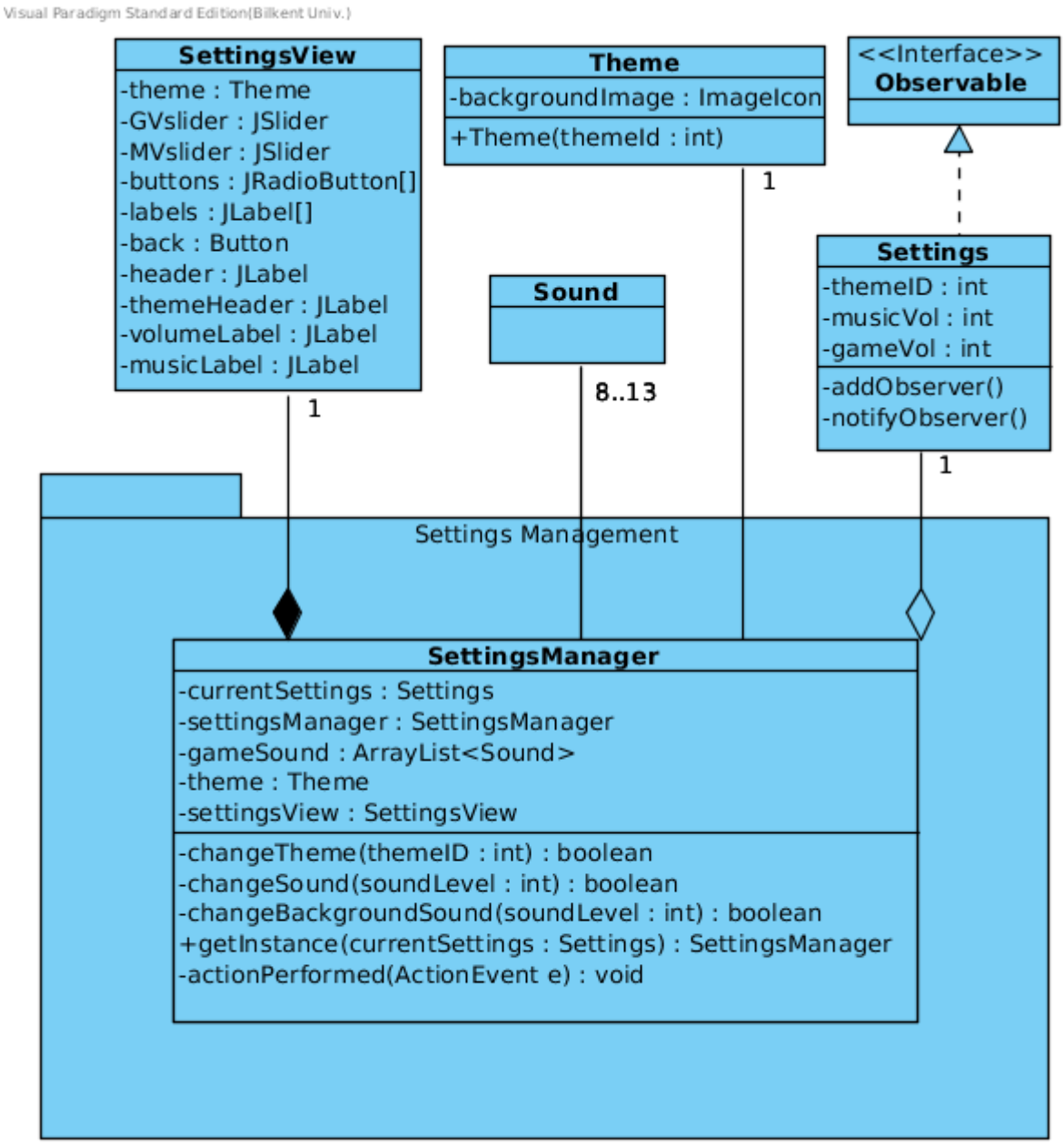

Figure 5: UML diagram of SettingsManager class

These are the attributes that SettingsManager will hold:

- settingsManager : SettingsManager - an instance of itself to enforce the singleton pattern

- currentSettings : Settings - an instance of the model Settings class which holds the data of the theme choice and the volume data.

- gameSound : ArrayList<Sound> - a list of all the sounds of the different objects in the game

-theme : Theme - an instance of the current theme of the game

-settingsView : SettingsView - an instance of the SettingsView from the buttons of which it will listen and for events and take the user input and update the Settings class, sounds and theme

accordingly.

- SettingsManager() The constructor will be private in order to be able to implement the singleton pattern.

These are the methods that SettingsManager will hold:

- changeTheme(themeID: int) : boolean - a method to update the theme view

- changeSound(soundLevel: int) : boolean - a method to update the volume of the sounds of the objects

- changeBackgroundSound(soundLevel: int) : boolean - a method to update the volume of the general background sound

+ getInstance(settings:Settings) : SettingsManager - a method which will return a new instance of SettingsManager in case it does not exist and otherwise it will return the existing one.

- actionPerfomed(ActionEvent e) : void - a method which will be invoked when an action will occur, such as change of volume in the settings or change of theme and which will update the settings model accordingly.

4.3 Game Management Package

Game Manager is the main manager which manages several aspects of the game. As the previously described manager classes it uses the singleton pattern and implements ActionListeners. All the attributes and methods of this class will be described in detail.

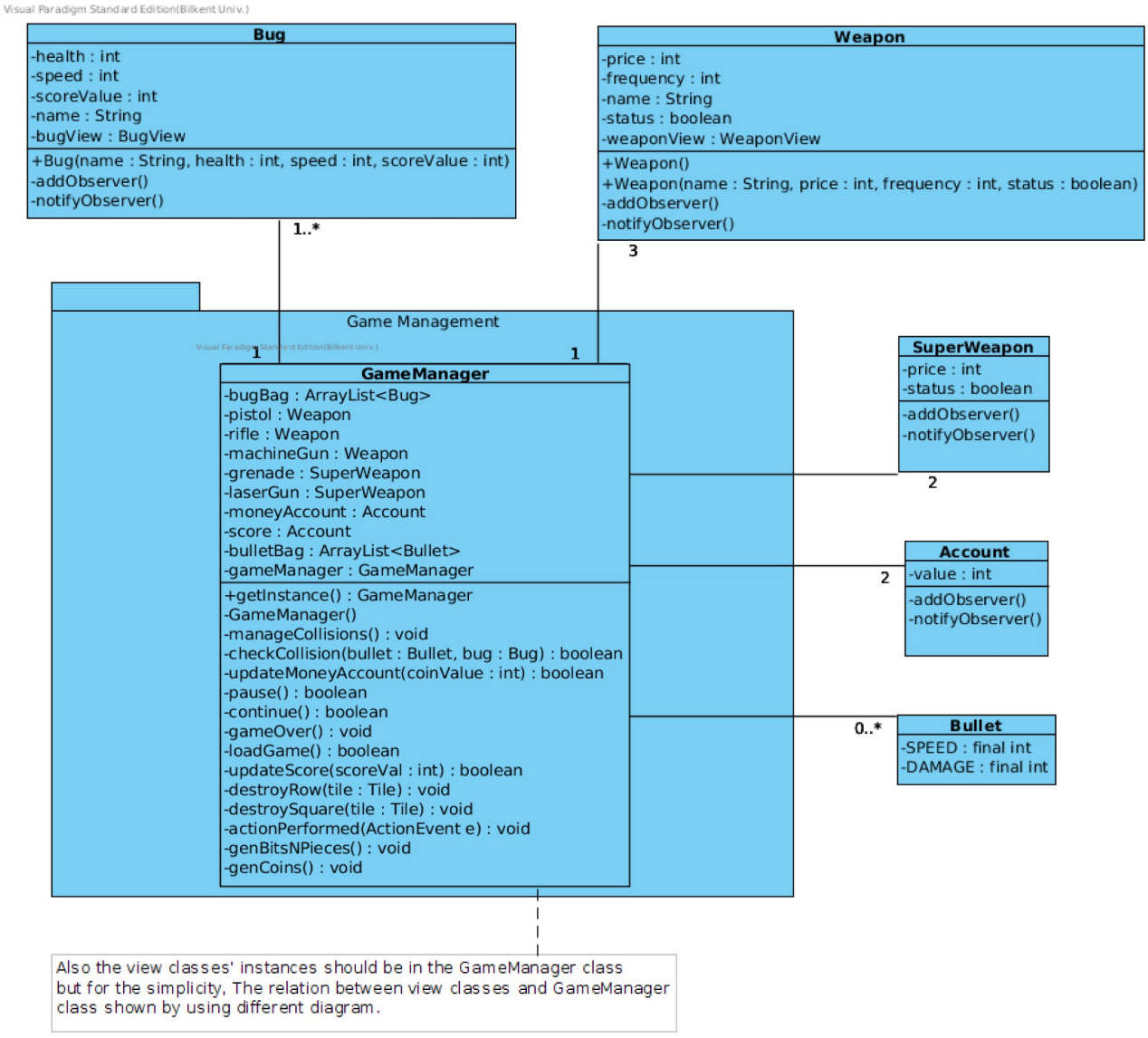

Figure 6: UML diagram of GameManager class

Attributes of GameManager:

- gameManager : GameManager - An instance of itself to enforce the singleton pattern.

- bugBag : ArrayList<Bug> - An ArrayList of bugs which will hold all the bugs which are currently in the game field. We decided to keep instances of all the bugs because we will need to check all the bugs for collisions with bullets.

- bulletBag : ArrayList<Bullet> - An ArrayList of bullets which will hold all the bullets which are currently in the game field. Similary, to the reasoning for the bugBag we decided to keep this list of bullets to be able to check for collisions with bugs

- pistol, rifle, machineGun - All weapon instances. We decided to keep one instance for which because there will be an instance of each in the game bar available or not available to be

bought. Thus according to the status of each of these weapons the price and availability will be displayed in the menu bar. In addition, we decided to not keep a list of the weapon objects which will be in the field being that after creation we don't need to keep track of the weapons which are in the field and the data they hold. The weapons in the field will be stationary and immutable. Thus a list of those weapons would be unnecessary.

- grenade, laser - Instances of Grenade and LaserGun respectively which are superweapons.

- moneyAccount, score : MoneyAccount - An instance of Account which holds the data of the money account and the score. When a coin will be collected or score will be gained, this class GameManager will update the data of this instance accordingly.

- views - It will have instances of main view classes to manage the main views which are MainMenuView, GameView, HighScoresView, SettingsView, HelpView, CreditsView. Additionaly in the MainInterface package, we have other view classes which are MoneyAccountTable, ScoreTable, Button, Theme. These view classes also manupulated by the game manager by getting the relevant information from the entity packages which are the model classes of the game. Moreover, for the game play management, the Game Management package have the attributes of the Game Field Interface package which are WeaponView, BulletView, GrenadeView, LaserGunView, Field, Tile, BitsNPiecesView, CoinView, GroceryView, BugView. Game Field package's view classes extends the GMWorld or GMActor classes according to their needs and roles into the game play screen.

- GameManager() The constructor will be private in order to be able to implement the singleton pattern.

The methods that this class is going to have are:

- loadGame() : boolean - This method will load the game with the default settings.

- checkCollision(bullet : Bullet,bug : Bug):boolean - This method will check if a bullet and a bug are colliding by checking the boundaries of both objects.

- manageCollisions() : void - This method will detect all the collisions that might happen and manage them accordingly. It will call the method checkCollision(Bullet, Bug) for each bullet and bug which are in the same row. We decided to implement collision check in this way in order to not do unnecessary checks as they will result in slow downs. Since bugs and bullets move only along the x-axis in the same row of the field a bullet and bug can collide only if they have the same x-coordinate. If there will be a collision between a bug and a bullet firstly the health of the bug being collided will be reduced according to the damage power of the bullet. Next, the health of the bug will be checked. If it is smaller or equal to zero the method updateMoneyAccount(bugValue:int) will be called.
In addition, this method will check if the bug has reached the end of the field If yes, the method gameOver() will be called (it will be described later in this section of the report).

- updateMoneyAccount(value : int) - This method will update the money account according

to a value provided as a parameter. This method will serve two purposes. It will update the money account when a coin is collected with the value of the coin as a parameter and it will also update it when a gun is bought with the negative of the price of the gun as a parameter.

- pause():void - This method will sleep the game.

- continue():void - It will wake up the game to the last playing state.

-gameOver():void - This method will display a pop up window and it will also create a File Manager which will then update and modify the high scores accordingly.

- updateScore(value : int) - This method will update the score of the Score instance when a bug is killed.

- destroyRow(tile : Tile) - This method will destroy all the bugs in the same row as the tile passed as parameter. This is a method of this class as only this class has access to all the bugs in the field and thus can make possible their destruction.

- destroySquare(tile: Tile) - Same as the previous method but destroys all bugs in a certain area around the tile passed as parameter.

- actionPerformed(ActionEvent e) - a method which will be invoked when an action will occur, such as when a button to buy a new gun is clicked, when the buttons pause/continue are clicked, when a tile is clicked to place a gun, when a coin is collected etc. The respective model instances are going to be updated accordingly. For instance, the MoneyAccount instance will be updated with the new value of the account.

- genBitsNPieces() - It is a method which randomly generates bitsNPieces views to be added to the field.

- genCoins() - This method randomly generates coins to be added to the field.

+ getInstance() - This method will return an instance of this class, GameManager in case it doesn’t already exist one and the existing one otherwise. This ensures singleton pattern, the creation of only one instance of a class.

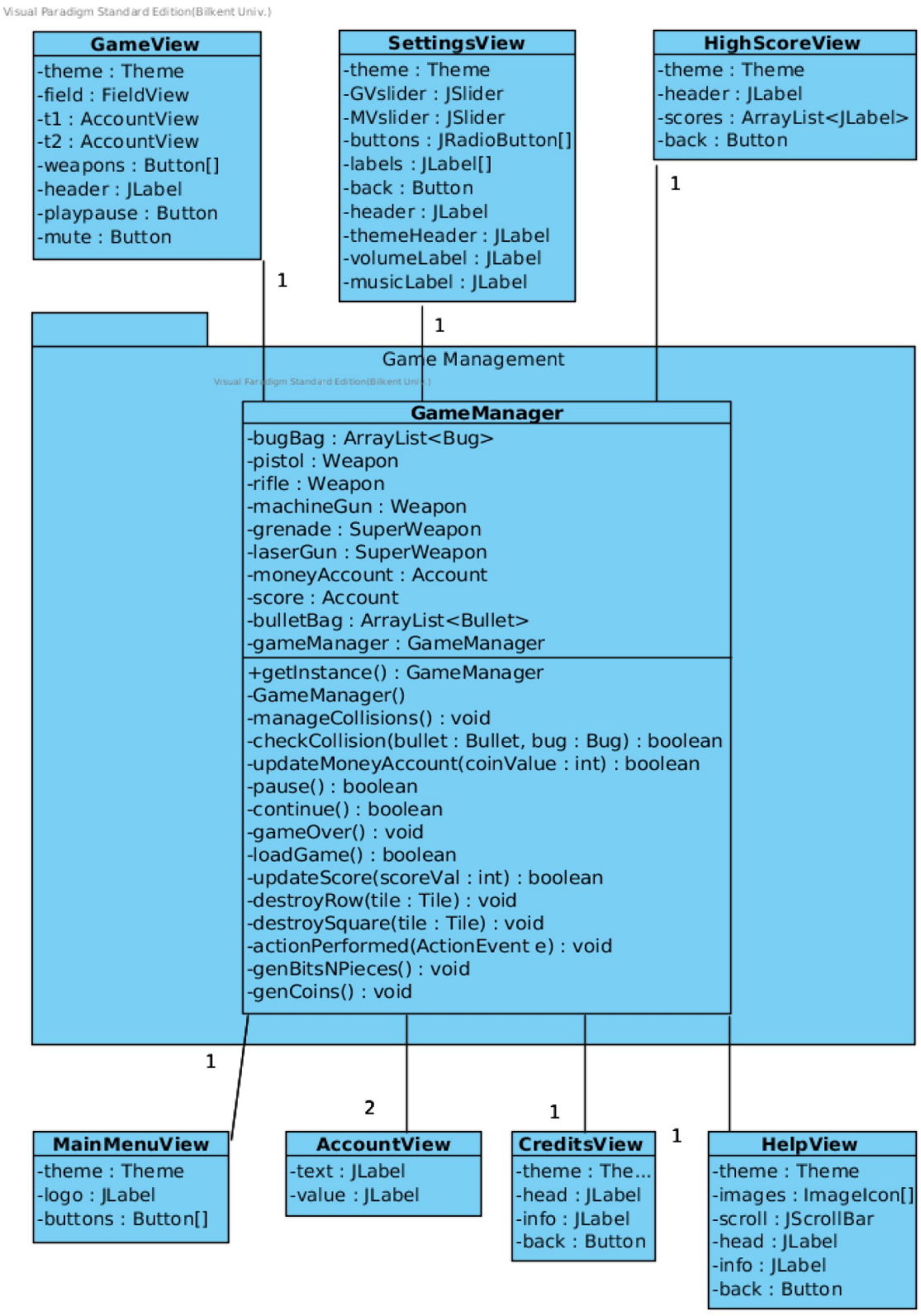

Figure 7: UML diagram of GameManager class with the view objects

4.4 Main Entities Package

This package composed of two classes which are the models classes of player and settings that holds the relevant data into these model classes,

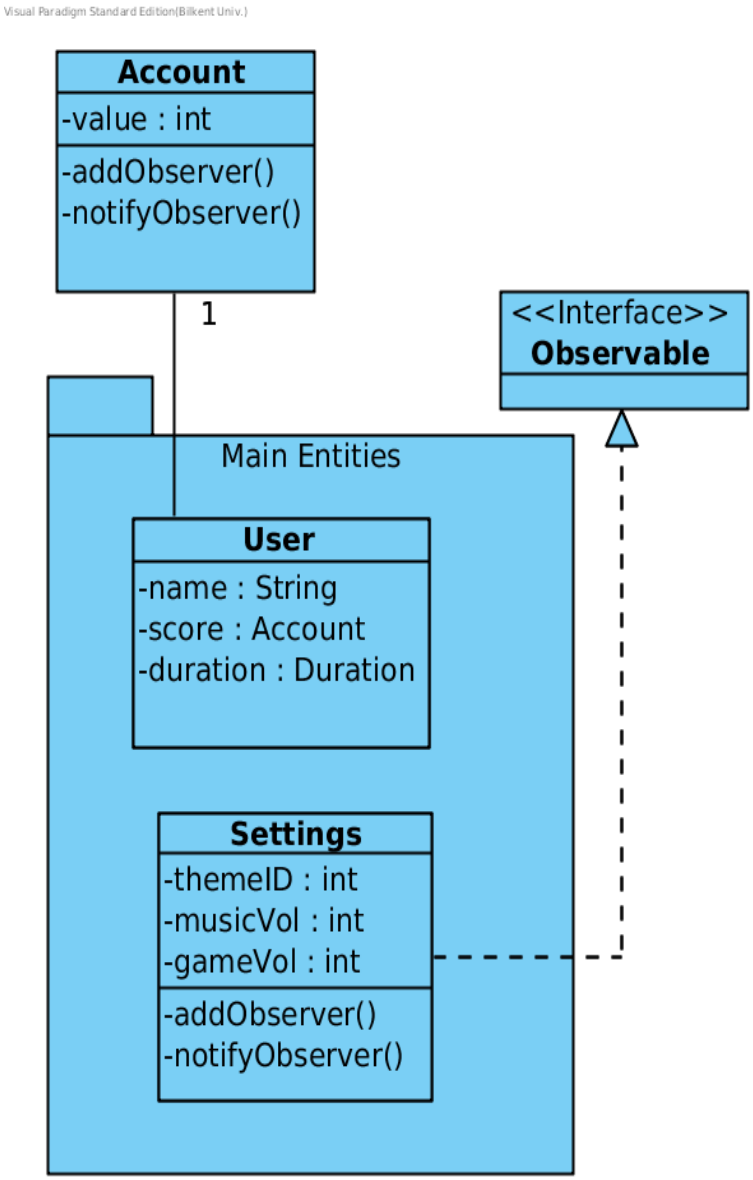

Figure 8: UML diagram of main entities and their relation with other classes

User Class

This class will hold some basic information for a user playing the game. It will hold the name, score and duration of the game that a user has played in the attributes name, score and duration respectively.

Settigs Class

This class holds information on the settings of the game. It will use the Observable - Observer pattern and as such Settings will extend Observable and its Observer will be the respective view

-themeID - an integer holding the id of the current theme of the game

-musicVol - an integer holding the volume of sounds of different objects of the game

-gameVol - an integer holding the volume of the background sound of the game

-settingsView - an instance of the SettingsView class which will be notified to change whenever there is a change in this class.

Methods that this class will have, despite getters and setters are the extended methods from Observable which are addObserver() and notifyObservers()

4.5 Game Entities Package

The Account class will serve for two purposes. There will be an instance for the money account and an instance for the score account. It has a variable named value which holds the value of the account. In addition to that it will also have a accountView instance which will be notified whenever the value variable is updated. This will be done by Account extending the Observable class and AccontView implementing the Observer interface.

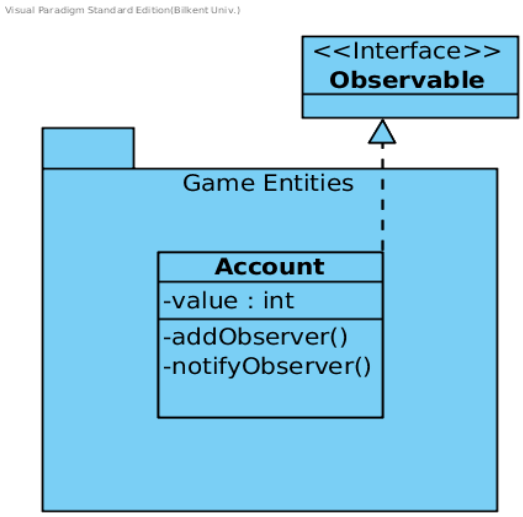

Figure 9: UML diagram of Account class

4.6 Game Field Entities Package

This package composed of the model classes of the actual game objects,

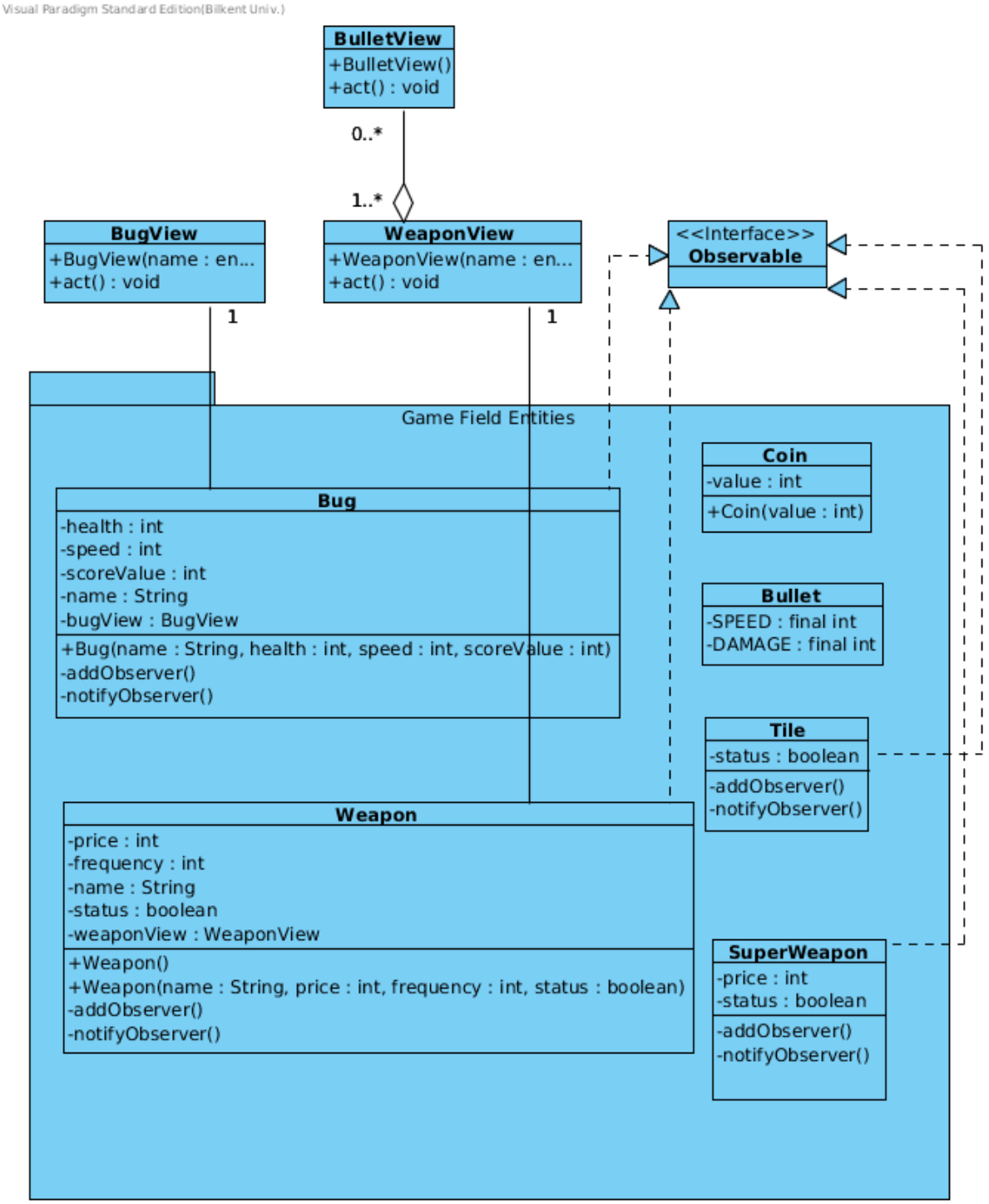

Figure 10: UML diagram of Game Field Entities package and their relations with the other neccesary classes

Weapon Class

Weapon is a class which holds information about a weapon object. It is an Observable class and its Observer is a WeaponView.
The attributes that this class has are:

-price - an integer holding the price of a weapon.

-frequency - an integer holding a frequency factor of the bullets coming out of a weapon per

a certain time.

-name - a name to distinguish between the different types of weapons.

-state - a boolean holding the status of a weapon, whether it is available to be bought or not.

-weaponView - a view which will construct a view for a weapon.

Methods of this class are addObserver() and notifyObservers()

SuperWeapon Class

The SuperWeapon class will serve as two types of weapons which are LaserGun and Grenade. These class will have a name, price and status variable to save the name(can be either LaserGun or Grenade) the price and the availability status to be bought. In addition it is an Observable class and it has the methods addObserver() and notiyObserver()

Coin Class

The Coin class is a model class for a coin and it will only store the value of the coin.

Bullet Class

The Bullet class is a model class for a bullet which holds the information regarding the speed and the damage that it can do to a bug in the integer speed and integer damage variable accordingly.

Bug Class

The Bug class is a model class for a bullet which holds the information regarding a bug which are name,health, speed and scoreValue.

In addition, it extends the Observable class and it has the methods addObserver and notifyObserver.

Tile Class

The tile class will be a class representing a tile and holding a boolean status variable. This variable will hold the availability of a tile (whether there is already a weapon in the tile).

4.7 Game Map Utility Package

Game map utility package can be thought like an API of the game, it holds two classes which are GMWorld and GMActor these classes will be inherited by the view classes to get the apropriate specialities. Shortly, this package provides a simple framework to the game play screen. The classes will be explained following pages,

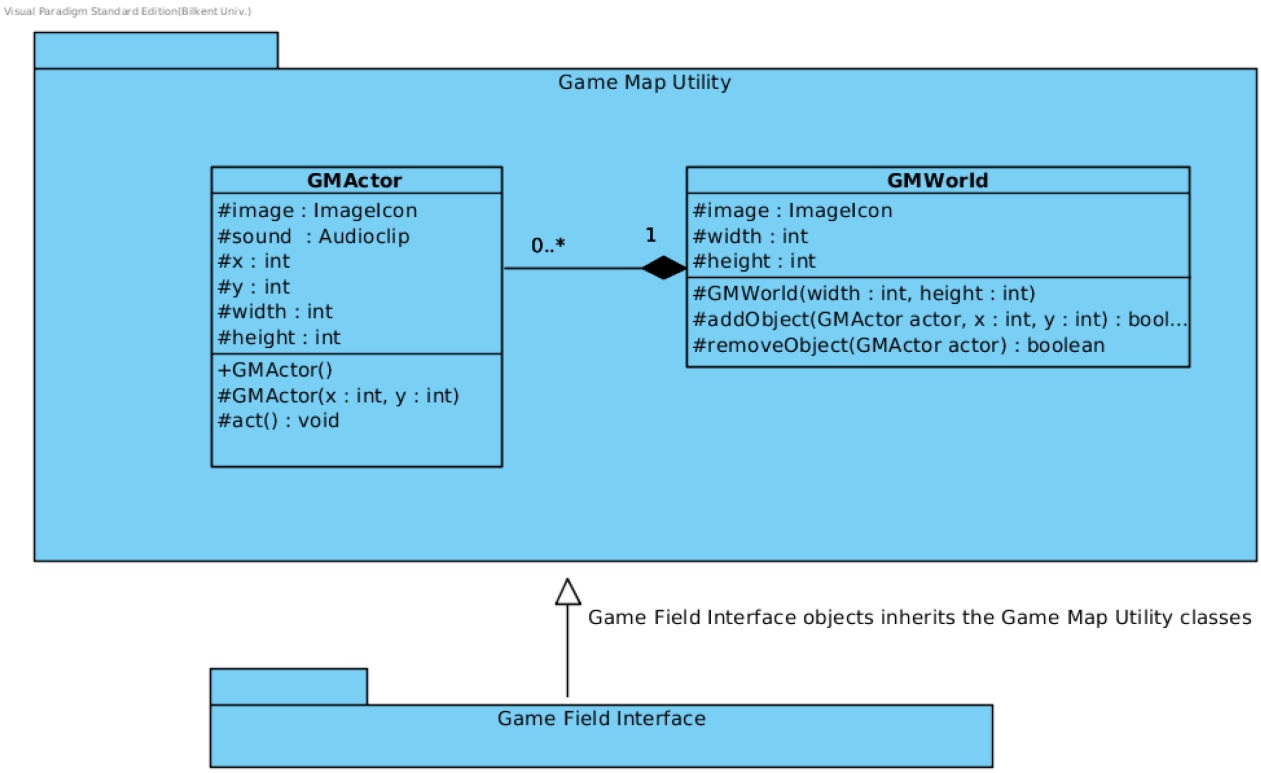

Figure 11 : UML diagram of Game Map Utility

GMWorld Class

GMWorld is a view class to do the game map utility design well. It initialize the first panel by determining width, height and image icon. It will add and remove all GMActor objects in the field so that GMWorld has a GMActor.

All instances are protected and only can be used by subclasses. Also, all instances have set and get methods.GMWorld holds these attributes;

#image : ImageIcon; instance of image to show the images of Actor by using imageIcon in png type

#width : int; instance keeps field's width which is final variable

#height : int; instance keeps field's height which is final variable

#GMWorld(width : int, heigth : int, background : ImageIcon)

#GMWorld(width : int, height : int); this constructor takes the width and height to draw final size of an field. It also keeps default image for the background.

#addObject(actor : GMActor , x : int, y : int) : boolean; this method adds specific Actor object to the field which is specified with x and y integer values for the new location. It is a boolean type to check the status of the availability which returns true/false

#removeObject(actor : GMActor) : boolean; this method removes specific Actor object from the field

GMActor Class

GMActor is a view class which represents the view of an object which is going to be placed in the field. It provides Actor objects' (bug, weapon etc.) functions and basic attributes via inheritance. In that way, the game field interface objects determine the images, sounds, coordinates etc. This class will be extended by view classes representing field objects which are in the Game Interface package.

All instances are protected and only can be used by subclasses. Also, all instances have set and get methods.GMActor holds these attributes;
#image : ImageIcon; instance of image to show the images of Actor by using imageIcon in png type

#sound Audioclip; instance of sound to give out a sound of Actor by using Audioclip in vaw type

int x, # int y; instances of x and y keeps the coordinate values of an Actor which can be changed for the location later

int width, # height; instances of width and height to determine of Actors' final size when actors are being using

#GMActor(); this constructor keeps default values for sound, image , x and y

#GMActor(x : int, y : int); this constructor takes integer x and y values as a parameter to determine location and updates them. These coordinates will be used by subclasses

#act() : void; this method unless overridden will not have any functionality.

4.8 Game Field Interface Package

All of Game Field Interface classes extends GMActor instead of GroceriesView and FieldView. Almost all methods uses extended GMActor methods but they have not be written here except overriding methods.

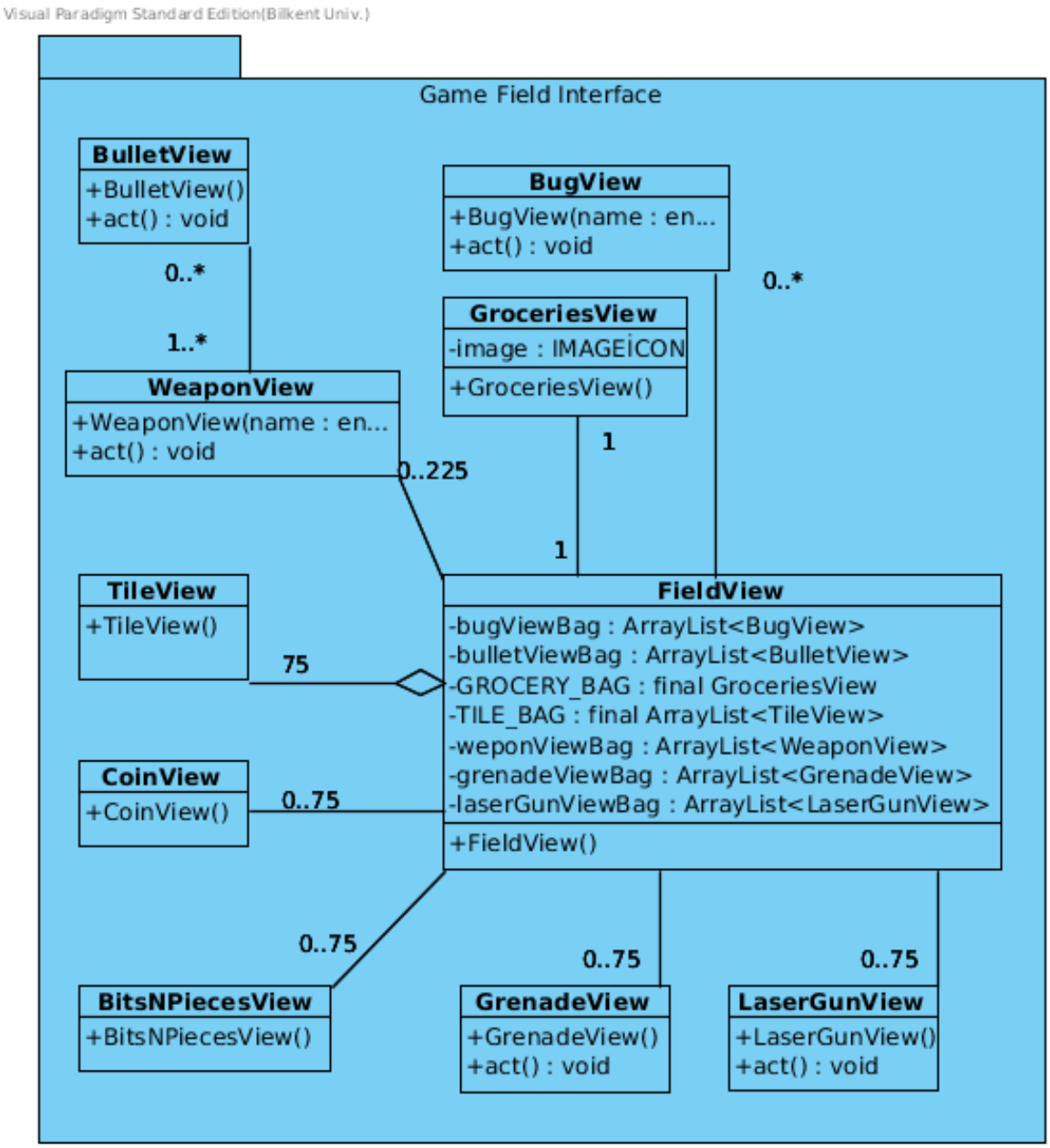

Figure 12: UML diagram of Game Field Package's classes and their relations with each other

WeaponView Class

WeaponView is a view class which inherits Observer interface to do the game field interface design appropriate as MVC pattern. It determines weapon objects' specific images and same type of acts depend on weapons' frequency level(rifle, machine etc.), in act method, it uses BulletView class to create bullets and fire them.

WeaponView uses necessary extended class's attributes.

+WeaponView(name : enum)

WeaponView uses necessary inherited method and holds these methods;

+WeaponView(name : enum); this constructor takes enum types of name as parameter and decides which image icon will be selected for the different weapon images

+act(); When this method is called, it creates bulletView objects depends on time period and fires these bulletView objects.

BitsNPiecesView Class

BitsNPiecesView is a view class. It provides BitsNPieces objects with specific 5 image for bits and pieces.

BitsNPiecesView uses necessary extended method and holds these methods;

+BitsNPiecesView(); this constructor provides images for bitsNPieces objects

BulletView Class

BulletView is a view class. BulletView provides Bullet objects specific one image into specific location and determine how to act after firing. BulletView is being used by the WeaponView to be thrown.

BulletView holds these methods;

+BulletView(); default constructor determining default image and sound of bulletView.

+act(); this method makes bullets move horizontally through the tile line

BugView Class

BugView is a view class which implements the Observer interface. It holds an image for the bug and has a method which makes a bug move across the GMWorld.

BugView uses necessary extended class's attributes.

BugView holds these methods;

+BugView(name : enum); this constructor takes enum types of name as parameter and decides accordingly which image icon will be selected for the certain bug.

+act(); this method is overridden from the GMActor class.

TileView Class

Tile is a view class which implements the Observer interface. will be appear and move on it. So FieldView class has TileView class.
Tile uses necessary extended class's attributes.

Tile holds this method;

+TileView() - This is the default constructor which creates specify the specific location of each tile cell on the field.

CoinView Class

CoinView is a view class representing a single coin and it holds the image of this coin.
CoinView constructor;

+CoinView(); this constructor specify the image of the coin.

LaserGunView Class

LaserGunView is a view class which implements the Observer interface. It is one of the two views of SuperWeapon model. Even if this superweapon looks similar as WeaponVew, it is different because it does not use BulletView class as an ammo. It uses its special one shot ammo and then supergun disappears. It holds an image of the laser gun object.
LaserGunView uses necessary extended method and holds this method;

+LaserGunView() - This is the default constructor which creates the LaserGunView object.

+act() : void - This method overrides GMActor's act method and fires as a the superweapon lasergun.

GrenadeView Class

GrenadeView is a view class which implements the Observer interface. This superweapon is similar to other super weapons but has different super ability, so it is also different from WeaponView. It uses its special one shot ammo and then supergun disappears. It provides image of the grenade object.
GrenadeView uses necessary extended method and holds this method;

+GrenadeView (); This is the default constructor which creates the GrenadeView object.

+act() : void - This method overrides GMActor's act method and fires as a the superweapon grenade..

GroceriesView Class

GroceriesView is a view class representing the groceries which are gonna be positioned at the end of the field.GroceriesView object have one specific image for groceries' reflection on the screen. GroceriesView holds this attribute;

-image : ImageIcon; this instance is for the image of the groceries

GroceriesView holds this method;

+GroceriesView(); this constructor determine image of groceries and its location.

FieldView Class

FieldView is a view class which extends the GMWorld class. This class holds all the GMActors which are in the field by keeping each kind in a separate list. It has a TileView and GroceriesView to prepare the field.

FieldView class' attributes;

-bugViewBag : ArrayList<BugView>

-bulletViewBag : ArrayList<BulletView>

-GROCERY_BAG : final GroceriesView

-TILE_BAG : final ArrayList<TileView>

-weponViewBag : ArrayList<WeaponView>

-grenadeViewBag : ArrayList<GrenadeView>

-laserGunViewBag : ArrayList<LaserGunView>

FieldView uses necessary extended method and holds this method;

+FieldView(); this constructor determines number of objects in the different arraylists.

4.9 Main Interface Package

Main Interface holds the essential visual parts for the main menu and its submenus.

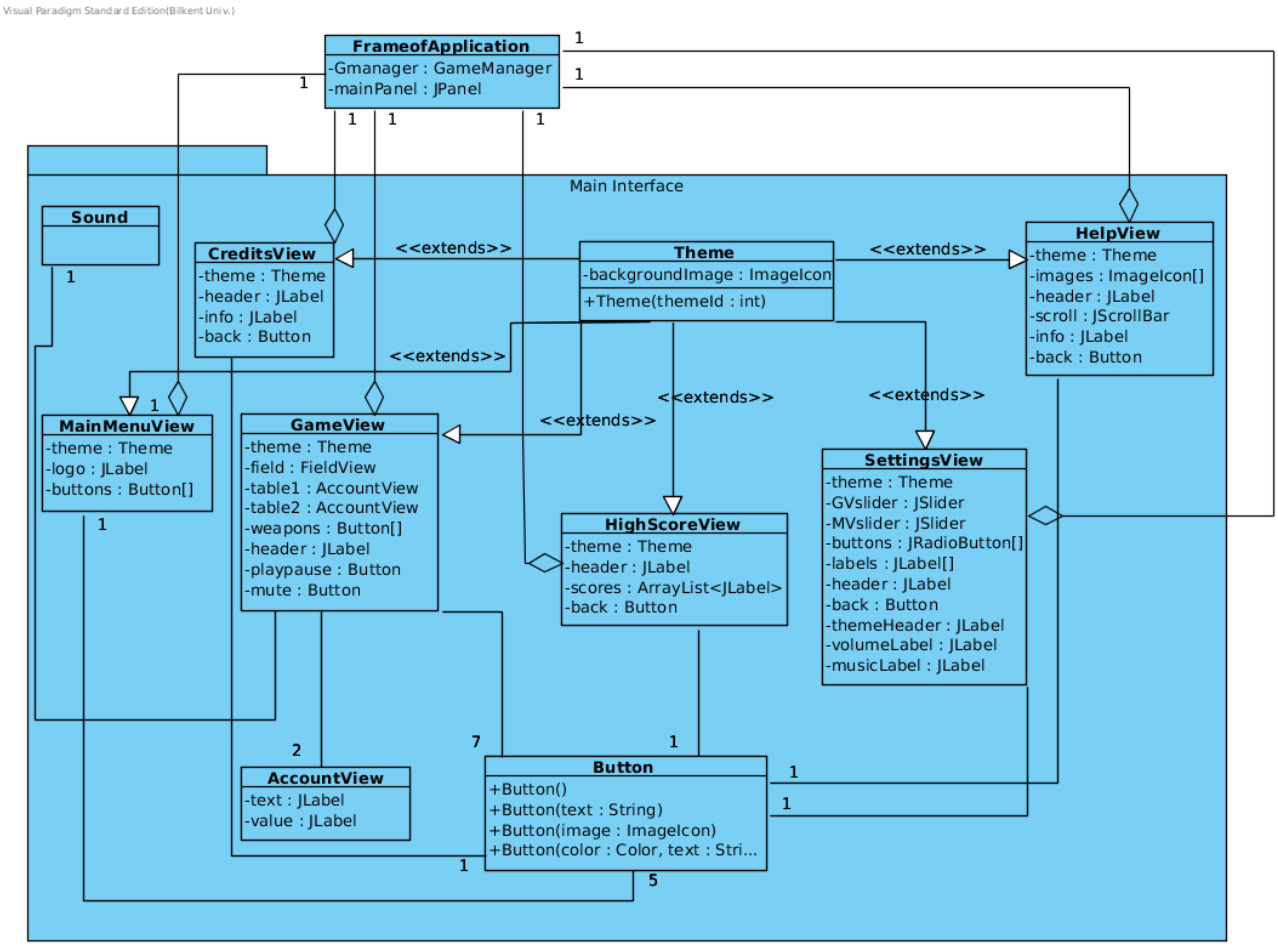

Figure 13 : UML diagram of main view classes

Button Class

Button is a customized class which extends JButton in order create buttons with customized features. We decided to create a Button class of ours in order to reduce duplicate code because many buttons will will have the same features and will all be customized from a JButton(color, font, shape).

It has several constructors.

+Button() - Default constructor of button class.

+Button(text : String) - Button constructor which takes string as a parameter.This constructor creates buttons with certain text on it.

+Button(image : ImageIcon) - Button constructor which takes imageIcon as a parameter. This creates a button with an image background.

+Button(color : Color, text : String) - Button constructor which takes a Color and a String as a parameter.This creates a button with a certain color background and a text in it.

Theme Class

Theme is a view class which creates different type of themes includes images according to user's decision. It extends JPanel, and GameView class will use this class.

Theme holds this attribute;

-backgroundImage : ImageIcon - This instance will be used for different images of themes

+Theme(themeId : int) - This constructor takes the themeID to determine different themes

AccountView Class

AccountView is a view class which represents both score table and money account table. It will be shown at the top of the game screen.

AccountView holds these attributes;

-text : JLabel - This will show the name of the labels("score" and "money account" texts only)

-value : JLabel - This will show the current value of score and money account separately

GameView Class

GameView is a view class which represents the main game view. It holds the FieldView, AccountViews, and different buttons etc. It extends JPanel to keep the other View objects in one main Panel.

GameView holds these attributes:

-theme : Theme - This instance holds the theme of the application.

-field : FieldView - This instance shows the tile and groceries.

-scoretable : Account - This instance shows the score table.

-moneyacttable : Account - This instance shows the money account table.

-weapons : Button[] - This array keeps the buttons in array and shows the different types of weapon selections.

-header : JLabel - This shows the game's name on top of screen.

-playpause : Button - This instance shows the play/pause button next to weapon buttons.

-mute : Button - This instance shows the mute button next to other buttons.

MainMenuView Class

MainMenuView is a view class which shows the game menu options. It shows the selection of the game options such as play game, settings, exit etc.

MainMenuView holds these attributes;

-theme : Theme - This shows the theme of the main entrance of the game

-logo : Jlabel - This shows the game's logo on top of screen

-buttons : Button[] - This button array keeps the all button selections

HighScoreView Class

HighScoreView is a view class which shows high score screen. It adds new high scores into the list and shows the updated list on the screen.

HighScoreView holds these attributes;

-theme : Theme - This shows the theme of the high score

-header : JLabel - It shows the headline of the high score

-scores : ArrayList<JLabel> - It keeps the score list in an arraylist and show it on the middle of the screen

-back : Button - Back button on right-bottom on the screen

SettingsView Class

SettingsView class is a view class which represents the Settings of the game.SettingsView holds these attributes;

-theme: Theme - This attribute holds the general theme of the application.

-GVSlider:JSlider - This slider button is for getting the information of the game volume level.

-MVSlider:JSlider - This slider button is for getting the information of background music volume level.

-buttons: JRadioButton[] - This attribute holds the array of radio buttons for getting the theme id. In that way user can select the theme that s/he wants.

-labels:JLabel [] - These labels will be used for defining the relevant theme names which are placed next to the radio buttons.

-header:JLabel - This attribute holds the header of the settings pageç

-themeHeader:JLabel - This attribute holds the theme header which will be placed just before the

theme selections.

-volumeLabel:Jlabel - This label attribute defines the relevant sliders process.

-musicLabel:Jlabel - This label attribute defines the relevant sliders process.

-back:Button - Back button will be placed left-bottom on the screen.

HelpView Class

HelpView class is a view class which shows the help screen to the user. It gives a quick tutorial about game play stage.

HelpView holds these attributes;

-theme: Theme - This attribute holds the general theme of the application.

-images: ImageIcon [] - This array holds the screenshots of the game to explain everythink to the user visually.

-header:Jlabel - This attribute holds the header information of the help page.

-scroll : JscrollBar - In the help view instead of using pages, there will be used a scrollbar to show the tutorial to the user in one page.

-info: Jlabel [] - This array holds the verbal explanation of the each screenshot.

-back:Button - Back button will be placed left-bottom on the screen.

CreditsView Class

CreditsView class is a view class which shows the credits screen to the user. It gives the basic information about the group members and contact information.

CreditsView holds these attributes;

-theme: Theme - This attribute holds the general theme of the application.

-header:Jlabel - This attribute holds the header information of the credits page.

-info : Jlabel - This attribute holds the information about group members and contact information.

-back:Button - Back button will be placed left-bottom on the screen.

5.) References

[1], [2], [6] Explanation of JRE and JVM, Oracle [Online]. Avaliable: https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqb/index.html, Accessed on: March 27,2016.

[3], [4], [8] Explanation of MVC Pattern, Wikipedia.[Online]. Avaliable: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller, Accessed on: March 27, 2016.

[5], [9] UML Tool. [Online]. Avaliable: https://www.lucidchart.com/, Accessed on: March 24,2016.

[7] MVC Pattern Diagram. [Online]. Avaliable: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#/media/File:MVC-Process.svg, Accessed on: March 25, 2016.