  
  
  
 **CS 319 - Object-Oriented Software Engineering  
Project Design Report**  
 **Break It!  
Group 11  
Nazlı Abaz**

**Caner Sezginer**

**Usama Moin**

Table of Contents

[Table of Contents 2](#_Toc466770234)

[1. Introduction](#_Toc466770235) 3

[1.1 Purpose of the System](#_Toc466770235) 3

[1.2 Design Goals](#_Toc466770235) 3

2. Software Architecture

[2.1. Overview 5](#_Toc466770236)

[2.2. Subsystem Decomposition 5](#_Toc466770237)

[2.3. Architectural Styles 8](#_Toc466770238)

[2.3.1 Layers 8](#_Toc466770239)

[2.4. Hardware / Software Mapping 9](#_Toc466770240)

[2.5. Persistent Data Management 9](#_Toc466770241)

[2.6. Access Control and Security 9](#_Toc466770242)

[2.7. Boundary Conditions 9](#_Toc466770243)

[3. Subsystem Services](#_Toc466770235) 10

[3.1.Detailed Object Design](#_Toc466770235) 10

[3.2. User Interface Management Subsystem](#_Toc466770235) 11

[3.3. Game Logic Subsystem](#_Toc466770235) 13

[3.4. Game Screen Elements Subsystem](#_Toc466770235) 15

[3.5. Enumerations](#_Toc466770235) 18

[3.6. Input Management Subsystem](#_Toc466770235) 19

1. **Introduction**
   1. **Purpose of the System**

Break It! is aimed to entertain and relax the player through a relatively basic gameplay. The easy to use user-friendly interface will allow the player to navigate the game without a hardship. The training part at the start of the game will enable the player to get into the main game with ease. As the levels get harder to play, the player will need to keep their focus and play the game without distractions. Our main purpose is to keep the player interested in the game throughout the levels.

* 1. **Design Goals**

**Performance:**

Response time of the game after the player has made a move is important. The game cannot lag as the game is dependent on the player escaping from attacks. If the response time of the moves made by the player is slower than the game’s automated attacks, then the game will not hold the interest of the player as they will keep dying.

**Dependability:**

The game should be able to detect which inputs are invalid or not. Invalid inputs should be ignored. The game cannot be open to fundamental changes made by the player as it can lead to the crash of the program.

**Maintenance:**

The game has to have extendibility to keep the player interested. The game has to be easy to add new levels and feature throughout the time. The game, also needs to be modifiable to be easier to fix any mistakes that were not detected at the time. By using Java, the game is easier to adapt to different platforms, there will platform independency.

**End User:**

The game has to be easy to use and learn to accomodate to any player of any age. This can be resolved by having a user-friendly user interface and creating tutorials and hints for the player. The player should be able modify the game in a way that suits their preferences.

**Trade-offs**

**Space – speed:**

As the game needs to have some considerable performance to run smoothly for the user, the memory space required may be increased depending on the speed up the game might need.

**Dependability – User Wants:**

To keep the game stable and without a crash or any lag, the game should not be tinkered with and some modifications should not be allowed. However, the player might want to change the game according to their wishes. There should be a fine line between what the player is allowed to change and inflexible enough to stabilize the game.

1. **Software Architecture**

## 2.1. Overview

The purpose of this section is the decomposition of system into maintainable subsystems. In order to reduce the dependency between different subsystems our approach would be to minimize the coupling between different subsystems as much as possible while trying to have high coherence between the classes in a sub system.

## 2.2. Subsystem Decomposition

In this section, the system is broken down into further systems in order to help understand how the system is actually going to function. The breaking down of the system is going to be an important factor as it is going to affect the overall performance of the game. We choose layered architecture because it suits the need of our project. Figure 1 showcases the three subsystems namely Game Engine, Game Mapping & Game Entities. The three subsystems would implement low coupling so that any changes in the future may not need the redesigning of the whole system. While identifying the subsystems the main objective was to design a system that can provide high cohesion accompanied with low coupling for example while looking at figure 2 we observe that classes with similar behavior are kept together in same subsystem.

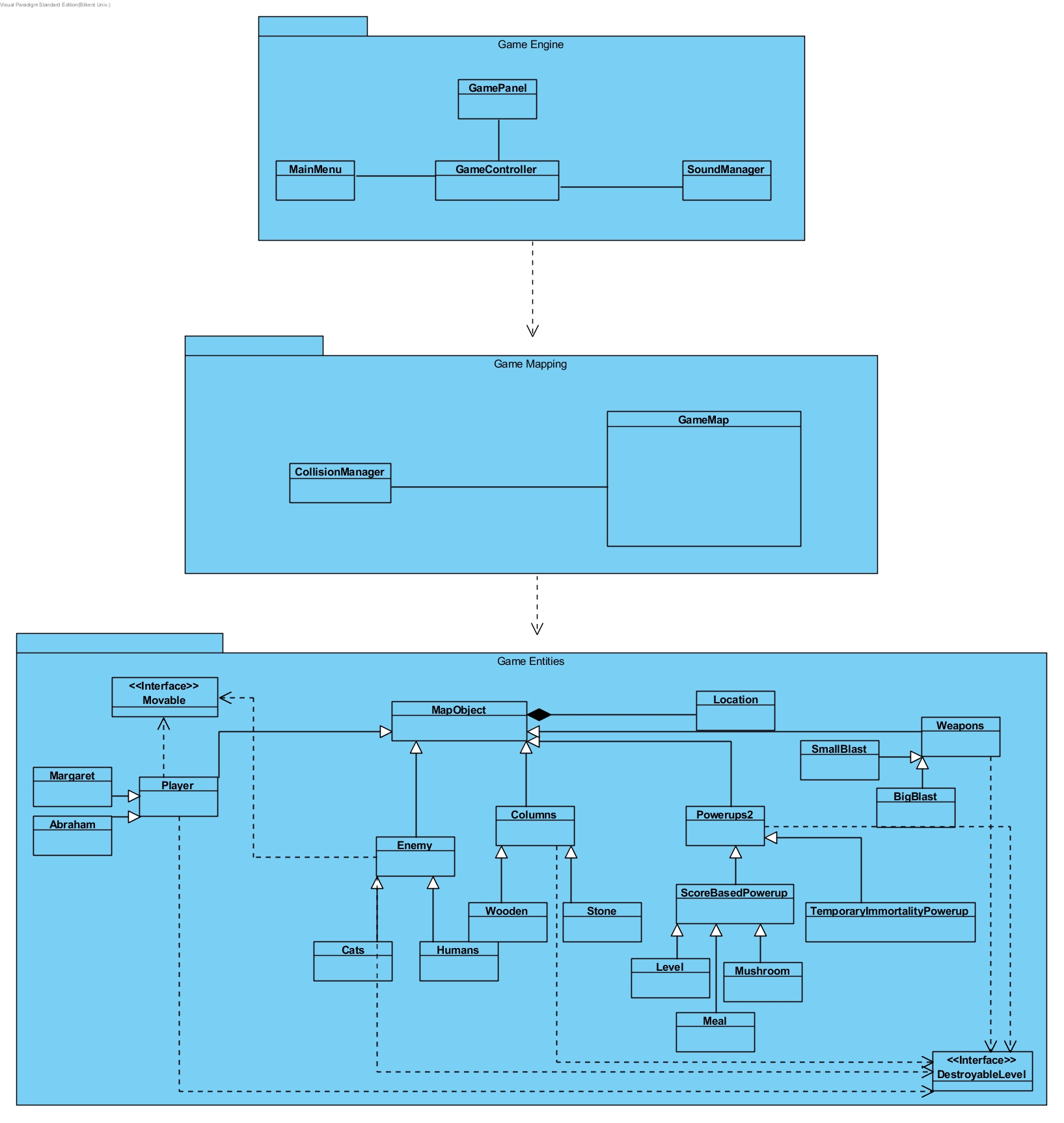


Figure 1 Basic Subsystem Decomposition

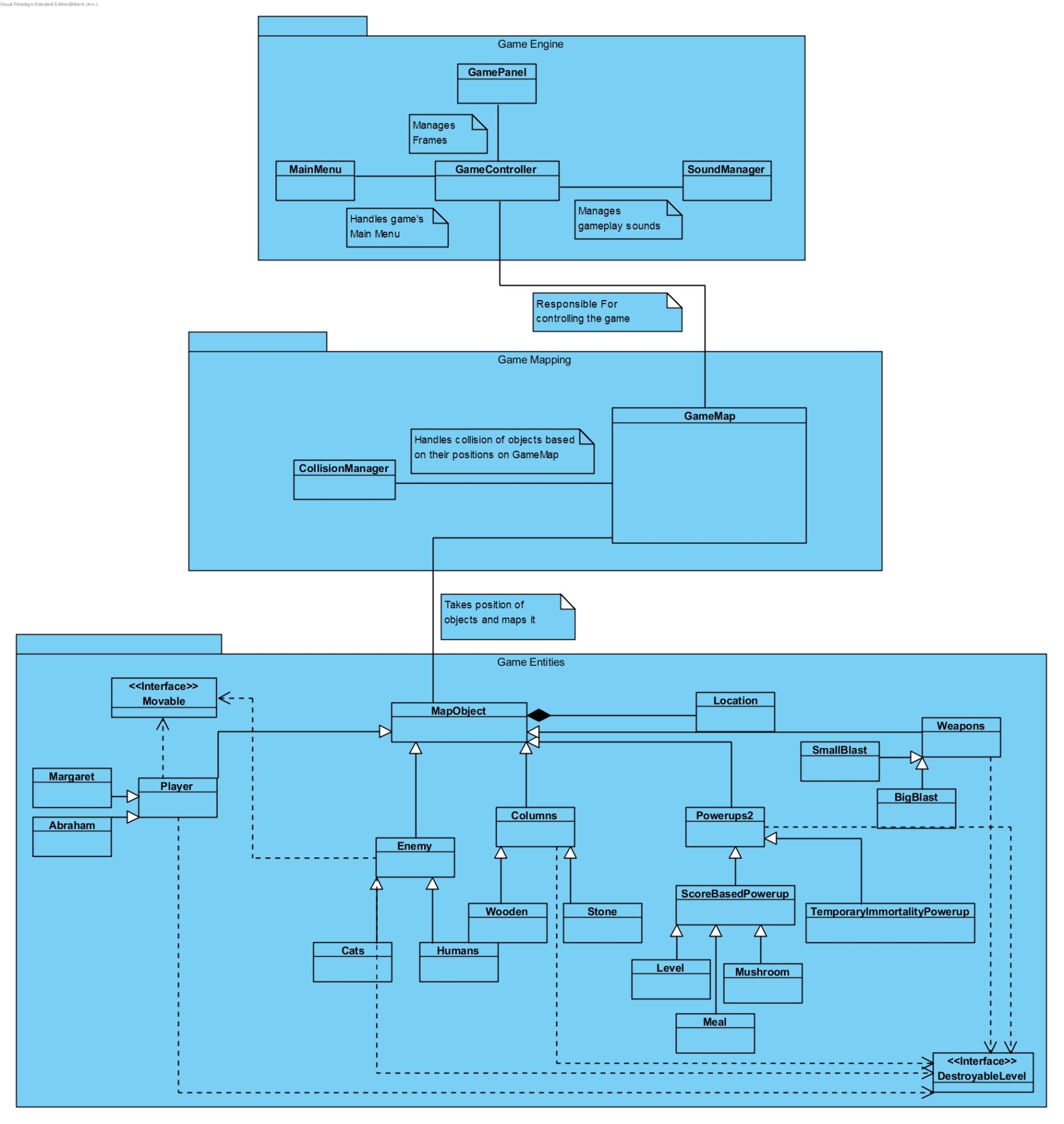


Figure 2 Detailed Subsytem Decomposition

## 

## 2.3. Architectural Styles

### 2.3.1 Layers

During the decomposition procees we broke down our system into three different layers namely Game Engine, Game Mapping & Game Entities. Subsystems have been made by combining related services and then hierarchy is created between those subsystems. The top most layer of the system is known as the Game Engine, this layer plays a major role initializing and handling the game, below that we have Game Mapping layer which is responsible for mapping all the objects on the game map as well as checking for collisions, after that we have Game Entities layer where we have all the necessary entity objects to be used throughout the game. Our layer decomposition follows closed architectural environment which means that a layer can only access the layer below it.

**Game Engine**

**Game Mapping**

**Game Entities**

Figure 3 Layers Of System

## 2.4. Hardware / Software Mapping

Break It! Will be developed using Java programming language therefore JRE(Java Runtime Environment) is required. For input the system requires keyboard only. The system requirements of the game are minimal as it just needs basic softwares and an operating system which are available in most of the systems these days. Input overhead of the system is very small since the number of keys it is supposed to handle is quite low.

## 2.5. Persistent Data Management

Game’s data would be stored in client’s hard drive, instead of database system we would be using text based storage. Where text files would be used for storing game maps,position information etc. Even if the text file gets corrupt during gameplay it wouldnt affect the gameplay but when the application is reopened it wouldn’t be able to execute with the corrupted file. The spritesheet with jpg format would be used in order to load spritesheet consisting of all the sprites, the sprites would then be extracted out from spritesheet.The sound files would also bes stored along with these files. No internet connection is required for the gameplay.

## 2.6. Access Control and Security

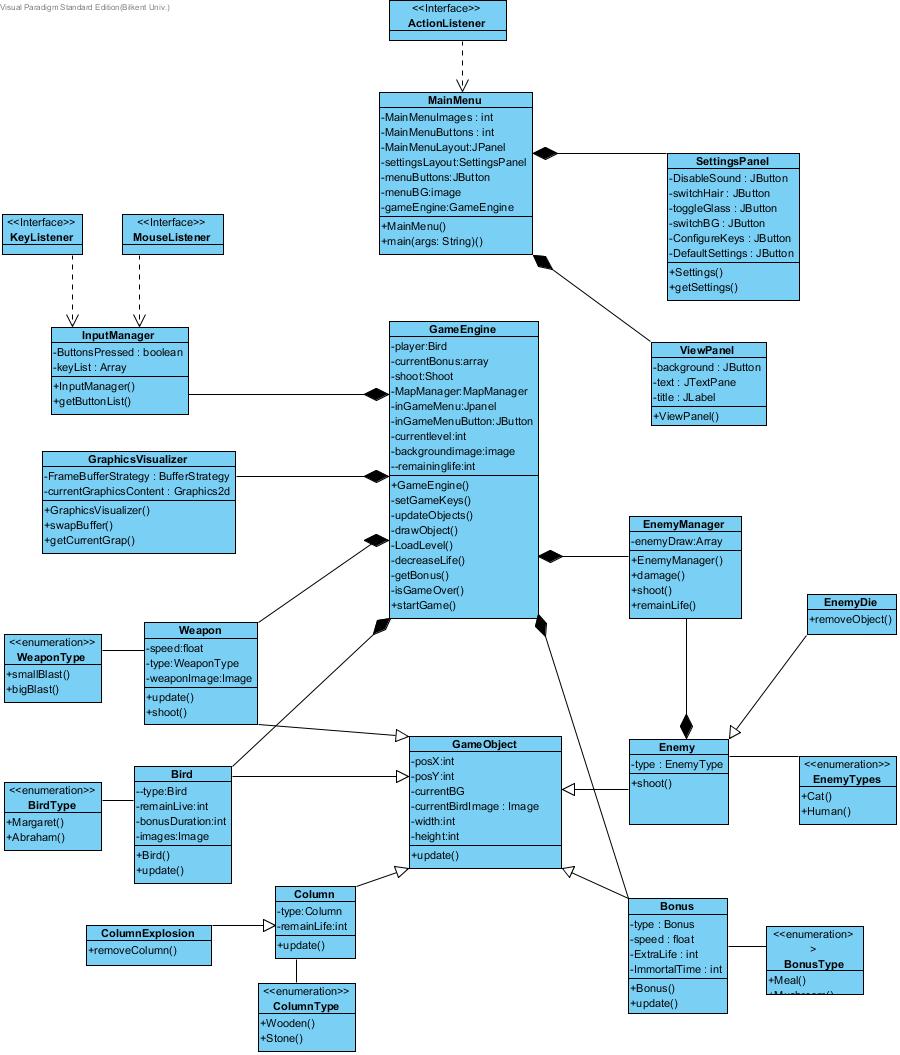
The game doesn’t require any user authentication therefore therefore no database is required to store user credentials. Moreover it wont be needing internet connection thus making it more secure.Anyone who executes the application have access to play the game so there are no restrictions for access. Therefore there wont be any kind of security issues faced in the application.

## 2.7. Boundary Conditions

The game doesnt require installation,at any moment user can exit the game by clicking the key specified for exit which would be displayed at the bottom of the screen,If any of the file is missing the game will stop executing and terminate itself.

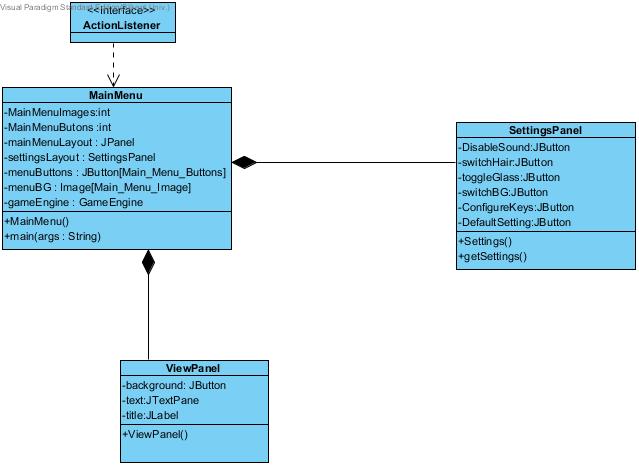
1. **Subsystem Services**
   1. **Detailed Object Design**

In order to provide a better understanding our project's interactions and basic fundamentals, we put the detailed class diagram below.

****

* 1. **User Interface Management Subsystem**

UIMS has three classes that are responsible to provide an interface between the user and the system.

****

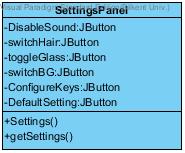
“ViewPanel” and “SettingsPanel” is used to display other submenus such as “Settings” and "Help". The “MainMenu” control to these classes when the user wants to open this options.

**Main Menu Class**

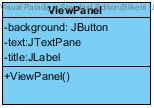
****

**Main Menu Class**

“MainMenu" will run firstly when the game is executed and displays the main menu. In this menu, there are four buttons such as “Play Game”, “Change Settings”, “View Help” and “Train Margaret”.

****

**SettingPanel Class**

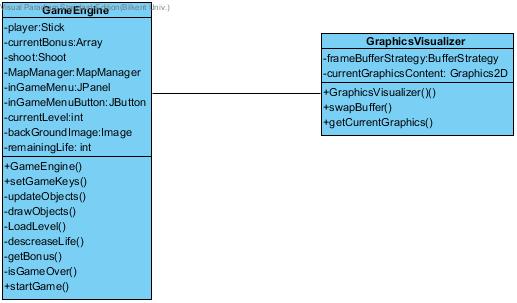
****

**ViewPanel Class**

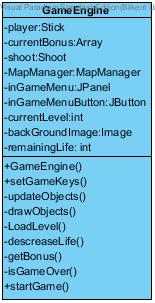
These are the other boundary classes of the UIMS which complements the “MainMenu” to ease its workload.

* 1. **Game Logic Subsystem**

Game Logic is responsible for mechanics of game such as destroy and display. Display management is covered by “GraphicsVisualizer”, the other gameplay mechanics are covered by “GameEngine” class.

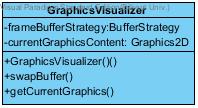
****

**GameEngine Class**

****

“GameEngine” is the main class of the entire system; that is why it covers almost everything about game via using its methods. Moreover, “GameEngine” class starts the main game loop and inside the loop, “GameEngine” class create the Bird, Enemies and the Columns according to level of game with its updateObjects() method;. Finally, at the end of the loop cycle it draws the objects to the screen using “GameVisualizer” class. GameEngine Class 16 game end conditions using isGameOver() method. If the isGameOver() methods return is true. It terminates the game loop and passes the flow of control back to the MainMenu class.

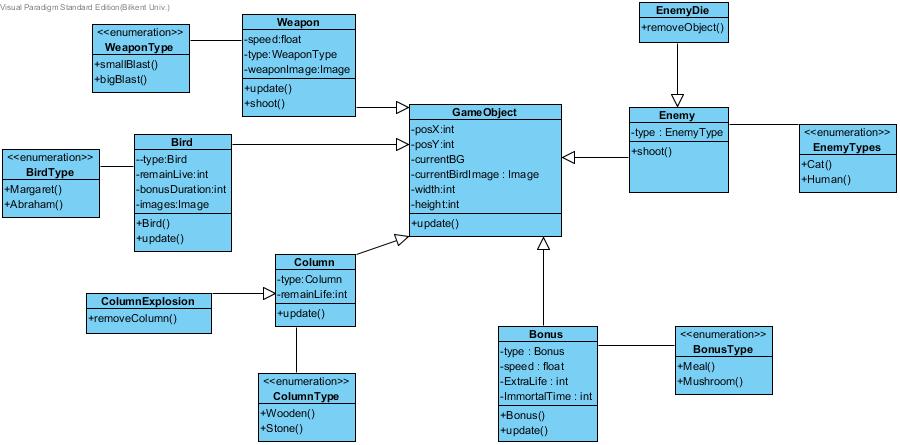
**GraphicsVisualizer Class**

****

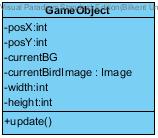
“GraphicsVisualizer” class handles the rendering operations of the game logic. It will be used by the “GameEngine” class in order to render the scene. Since the program will use double buffering technique, “GraphicsVisualizer” class designed to handle buffer swaps using the Java® “BufferStrategy” class. While we rendering the current game elements to the buffer, we display the previously rendered buffer.

* 1. **Game Screen Elements Subsystem**

“Game Screen Elements Subsystem” declares the objects to show in the screen while the game is actually running. It has many crucial objects such as “Bird”, “Column”, “Weapon”, “Bonus” and “GameObject”.

****

**GameObject Class**

****

“GameObject” will be instantiated after user decides to play the game. All fundamental objects use the “GameObject” abstract class as a parent class since they all need to have position information such as “PosX” and “PosY”, size information such as “width” and “height”. Therefore “GameObject” has all these attributes. We require to draw all of the objects on to the screen therefore “GameObject” class has a image that will be drawn by the “GraphicsVisualizer” class.

**Bird Class**

****

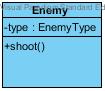
The “Bird” class has additional attributes in addition to the GameObject class’s attributes, such as “speedX” and “speedY” because it needs to aim through X and Y directions. Also it stores the state of bird by using enumeration, which will be overwritten when a new state is active. It also includes the implementation of “update()” method. The “update()” method manages the position of “Bird” according to its speed.

**Column Class**

****

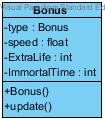
The “Column” class stores remaining number of lifes which will be decreased after shooting by bird.

**Enemy Class**

****

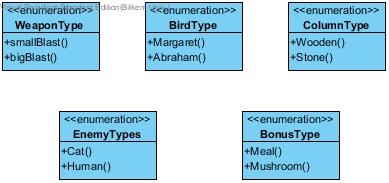
The “Enemy” class stores type of enemy and also has a method called as shoot .

**Bonus Class**

****

The “Bonus” class stores type of bonus and their effects on the game .

* 1. **Enumerations (Stick State, Ball State, Bonus Type Enumerations)**

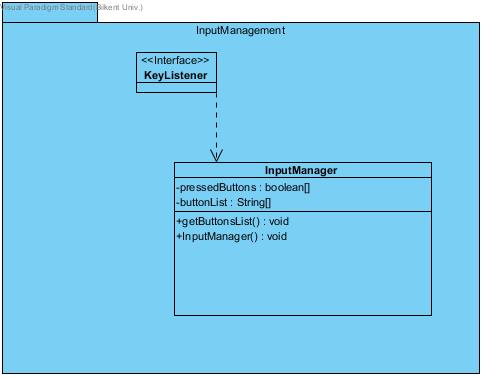
****

We used five different enumerations in “Game Screen Elements Subsystem” such as “WeaponType”, “EnemyType” , “BonusType” ,“BirdType” and “ColumnType” since we have a lot of states and types. Instead of using all these types as integers we use these enumerations because they will ease our implementation organization.

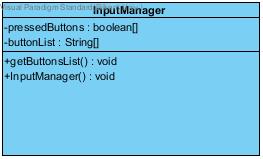
* 1. **Input Management Subsystem**

This subsystem handles all the inputs the system requires to run properly. The player

plays the game through commands made with the inputs.



**Input Manager Class**



“InputManager” is the class that enables keyboard usage to the player. “pressedButtons” keeps which buttons are being pressed by the player. “buttonList” is the array of all the buttons that can be activated throughout the game. “getButtonsList” is used by the game engine to find the buttons used at the moment.