CS 319
Object-Oriented Software Engineering Project
Design Report

Section 1 / Group 1H

EceAltınyollar
Arif Can Terzioğlu
Raza Faraz
EminTosun

Instructor: Bora Güngören

TABLE OF CONTENTS

1) Design Goals

1.1) User

1.1.1) Well-Defined Interface

User interface is one of the most important thing for a game to make it desirable for the users. Because of this reason we planned for Battle for Middle Earth to be provided with simple interactive objects. Buttons locations and the description of buttons are going to be clear and understandable for the user. Also the clear guidance is going to be provided for the comfort of players. The backgrounds of the screens will be the Middle Earth theme. Also while the players play Battle for Middle Earth, they will have characters and enemies with middle earth theme. These features make the game more realistic and desirable for the players.

1.1.2) Ease of Use/ Learn

Easiness of usage and learning a program to a user is one of the purpose for creators. Since the users do not want such a programs which have difficult usage and hard to learn how to use. For the usage purpose the screens that user has to pass over are going to visualized before the game starts. The users can do easily the required parts to start the game. Also the help section will be available for learning how to play Battle for Middle Earth. In the help section control of the character, fire controller and power-ups will be explained. With these features we are going to reach this purpose.

1.1.3) Performance

The performance of a program affects the quality of a game. So we will make Battle for Middle Earth as a high performance game. First of all the passes between the screens will be smooth. Because of this is an interactive program the game also will have response time as short as possible. With these Battle for Middle Earth will work nicely without any glitches. These features make the game more qualified.

1.2) Maintenance

1.2.1) Understandability

Each part of our project will be understandable for anyone who uses and analyses our project. In order to provide this opportunity to people we design our reports very simple and clearly. We also focus on completeness of the project. The coding part also will be clear and much more commented for increasing project's understandability.

1.2.2) Modifiability

When project needs to be modified, change in one part should not affect other parts of the program. In order to achieve this goal, we need to keep coupling at the minimum level, so that change in one part of the code, the whole project would not be affected too much. We will have some subsystems. These subsystems will share less data because modifiability of the project could be higher.

1.2.3) Good Documentation

We will keep our documentation standards as high as we can. By providing good documentation, we make our project more understandable and more clear. The completeness of reports and their relations with each other will be taking into consideration more seriously. This will also increase our projects understandability.

1.2.4) Portability

Since there are many different types of devices, operating systems and versions, our program may encounter such a problem that platform dependence or support issues. To ensure portability and have a stable performance between these environments, our development language will be JAVA which is supported widely by MAC OS, WINDOWS and LINUX… Thus, game will be played for most of the systems that can run JVM.

Also, by using SQL, our database will be managed between relational database management systems.

1.2.5) Reliability

Our game will be produced with the principle of reliability. To achieve this, it will be ensured that program will not be crushed under any users' input. Possible error cases will be determined during the design stage. To handle these errors, exceptions and user based solutions (giving error message and showing options to user) will be used. The exceptions will then redirect the program to normal operating way.

1.3) Trade off

1.3.1) Portability vs Reliability

To achieve the portability goals, we will use JAVA environment for implementation. JAVA has its own bytecodes which is known for its ability to produce byte codes which can run on any processor architecture. However, this byte codes only work in JAVA Virtual Machine (JVM). Thus, all users have to install JAVA to their computer. On the other hand, compared to pure compilation languages, JAVA does not have a good running time performance as much as them. Due to game is not complex too much; this trade-off will not be too much problem.
By using JAVA, due to compilation time checks, our software will be aimed error free as much as possible.

1.3.2) Development Time vs Performance

To decrease the development time of this game, the implementation project will be in java. The reason being Java offers much better and readily available GUI libraries, specific to our needs. Additionally we do not have to worry about possible memory leaks, which will not only decrease the development time of the game ,but in a way increase the performance of the game for example if inefficient memory management happens the game might store garbage or redundant values which will increase the space taken during the game, thus reducing the game performance.

2) Software Architecture

2.1) Overview

In this section, we are detailing the composition of our system from higher level to lower. While dividing our program, we decided to use 3-tier design architecture because it is suitable for our project. The layers will be explained in this section but more details of subsystems will be in other sections.

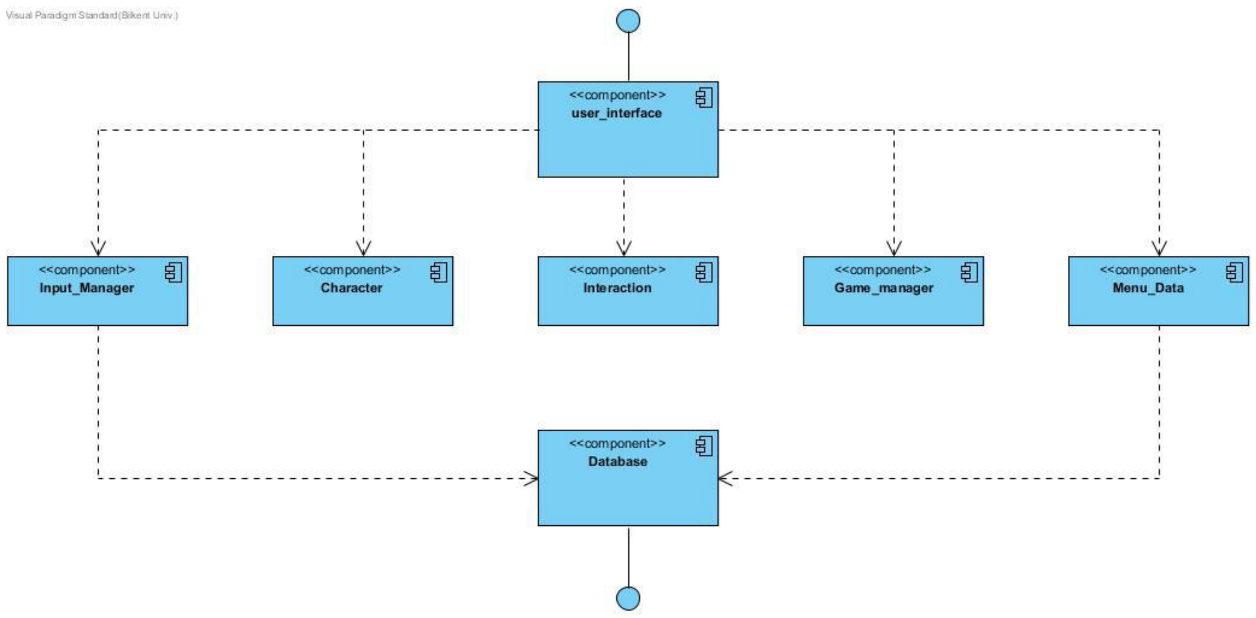
2.2) Architecture Style

We have chosen the three tier architecture style for our project. In this style, program will be divided into three layers. First layer is presentation layer. The presentation layer is responsible for managing the interaction between user and program. It shows graphics to the user with data coming from program and it also transmits the user's inputs to the program. The second layer of the architecture is application layer. Its duty is creating the applications data and process the data coming from user or database. The last layer is data layer. This layer is responsible for data management. It transmits data which needs to be stored to the database and it also conveys the data from database to the program itself.

2.2.1) Type of Architecture

For this project, we chose the closed architecture instead of open architecture. The close architecture minimizes the coupling. It makes easier to develop the game. Adding new things to the game will not require to change whole code of it. We can add new things by changing only related parts.

2.3) Subsystem Decomposition



2.3.1)First Layer

In the first layer, we have UI classes and UI Manager class. This layer is responsible for managing interaction with user and the program. It will get data from second layer which handles the business logic and it will be updated with new coming data.

2.3.2)Second Layer

The second layer of architecture is responsible for program's business logic. It will produce game data and send it to first layer which handles the interaction with user. This layer also manages data coming from database. This layer is like a bridge between database and UI. Whole game data processing will be handled with this layer's components.

2.3.3)Third Layer

Third layer of architecture is data management layer. This layer has the database controller component. This component provides data, which is coming from database, to the second layer of architecture which uses this data. It is also responsible for saving data coming from program itself or user.

2.4) Hardware / Software Management

Our game BattleForMiddleEarth will be coded in JAVA environment since JAVA 8 provides us for beneficial libraries such JAVA FX GUI library. Also, account information and high scores will be saved via MySQL. Storing images and sounds will be in the .png and .wav formats respectively, the operating system should support them.

On the hardware configuration part, game will need a basic keyboard controls that are space and arrow keys. Our game will not require too much system specification such as high RAM etc. In addition, at the beginning of the game, keyboard inputs will be necessary for create or login account.

Any platform such as Windows/macOS/Linux supports the JAVA can run our game with these hardware and software requirements.

2.5) Persistent Data Management

External database will be used to hold user's latest level, high scores, account information and user's preferences like character type. To load the latest point of players in the game, this information will be taken later.
Changes on data will be reflecting immediately to maintain data flow fluently. Database will have MySQL management system and SQL language will be the used to create query for database operations. Other files such as pictures, sounds will be stored locally.

2.6) Access Control and Security

Our game will require a connection to database. The username and password will be taken when user enters the game first time and transfer to database. Although game has no internet connection, it is aimed that different accounts will enable user to play with more than one player without losing without other characters' data. In other words, access control will be used to differentiate user's characters. Our program will be coded in a way that only our program can access it.

2.7) Boundary conditions

2.7.1)Initialization

Battle for Middle earth will not require any installation as the game would be available as executable .JAR file. During the stage when the .JAR file is clicked and the program opens, appropriate Graphical Interface will be loaded such as game images etc ; thus the game would be brought to an initializing state.

2.7.2)Termination

There are multiple possible ways for the termination of the program. Standard procedure includes clicking on the 'Exit' button present in the 'Main Menu' Screen and 'Paused' game screen. Additional method includes clicking on the 'X' present at the top left corner of the screen of using Task Manager.

2.7.3)Failure

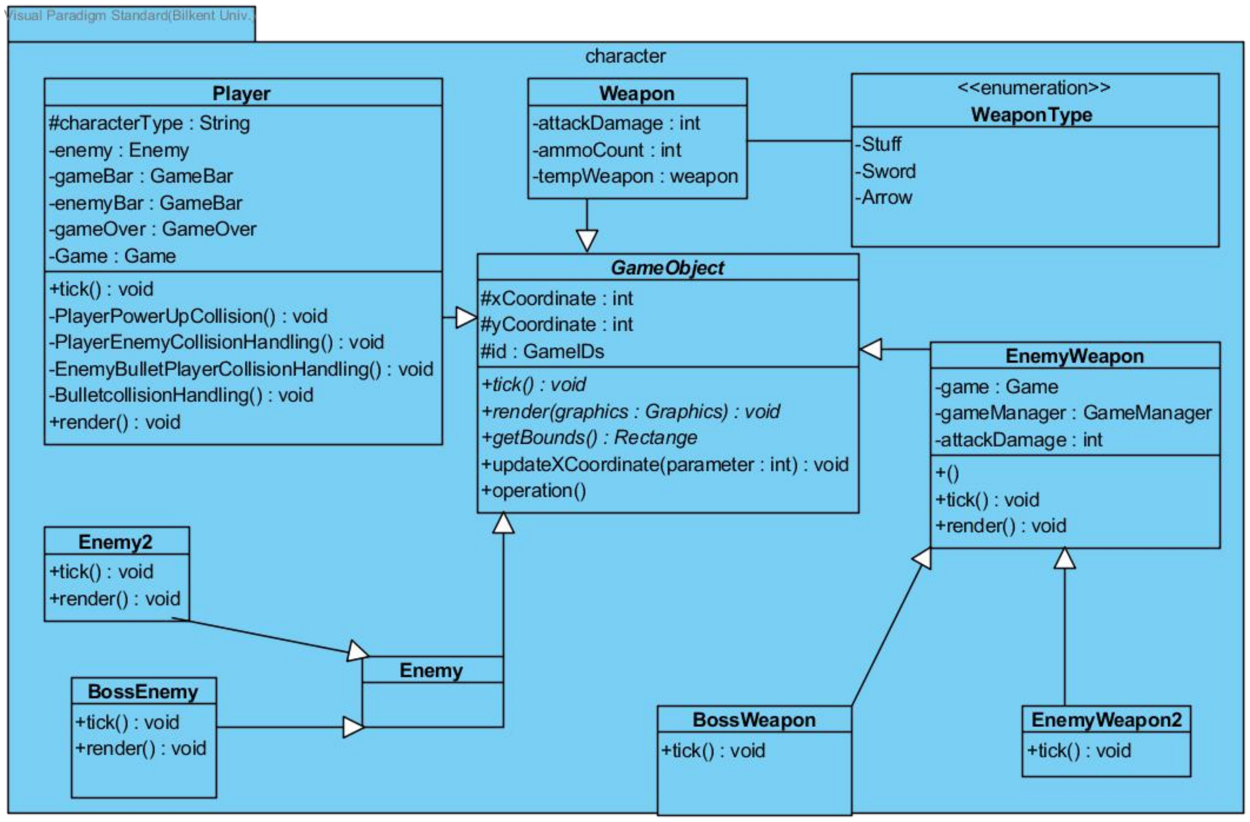
Possible Failure would result from missing program files in which a dedicated error message will be produced. Other Possible Failure includes absence of .JDK file or updated .JDK file which would cause the game to crash , thus displaying an appropriate error message.

3) Subsystem Services

3.1) Façade Design Pattern:

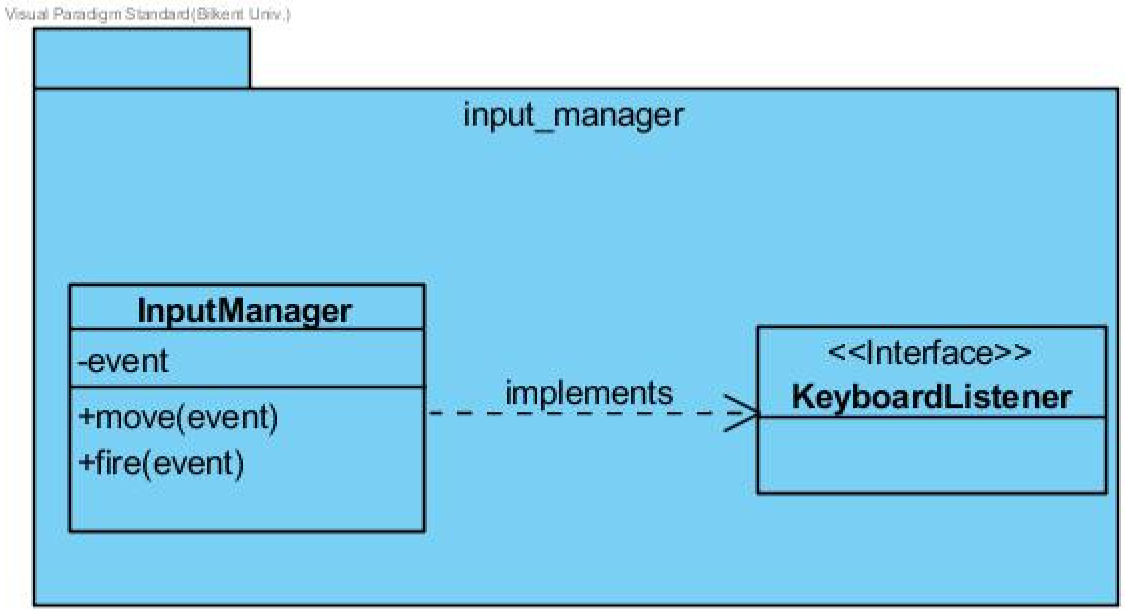
In our design pattern the Façade class is the 'Game Object' class, as this class handles most of the main functions of the game. By using this class as our façade class , it would be easier for future or current developers to maintain , reuse and extend the functionalities of the game. This is because most of the classes are linked with this class thus any alteration to this façade class will directly have an effect on the other components of this subsystem.

3.2) Character



The character component is in the second layer of our three-tier architecture. This component includes Player, Weapon classes , Enemy classes , GameObject class and weapon and enemy Weapon classes. The data about account and the features of characters that player choose is in this component. The character component interacts with interaction and database components. Link between Interaction component and character component provide the data about levels that account has. Character component takes the data that about the passed level and it provides data about current level to interaction component. The character component also provides the login information of the account to database component and takes the account information from the database component.

3.3) Input Manager

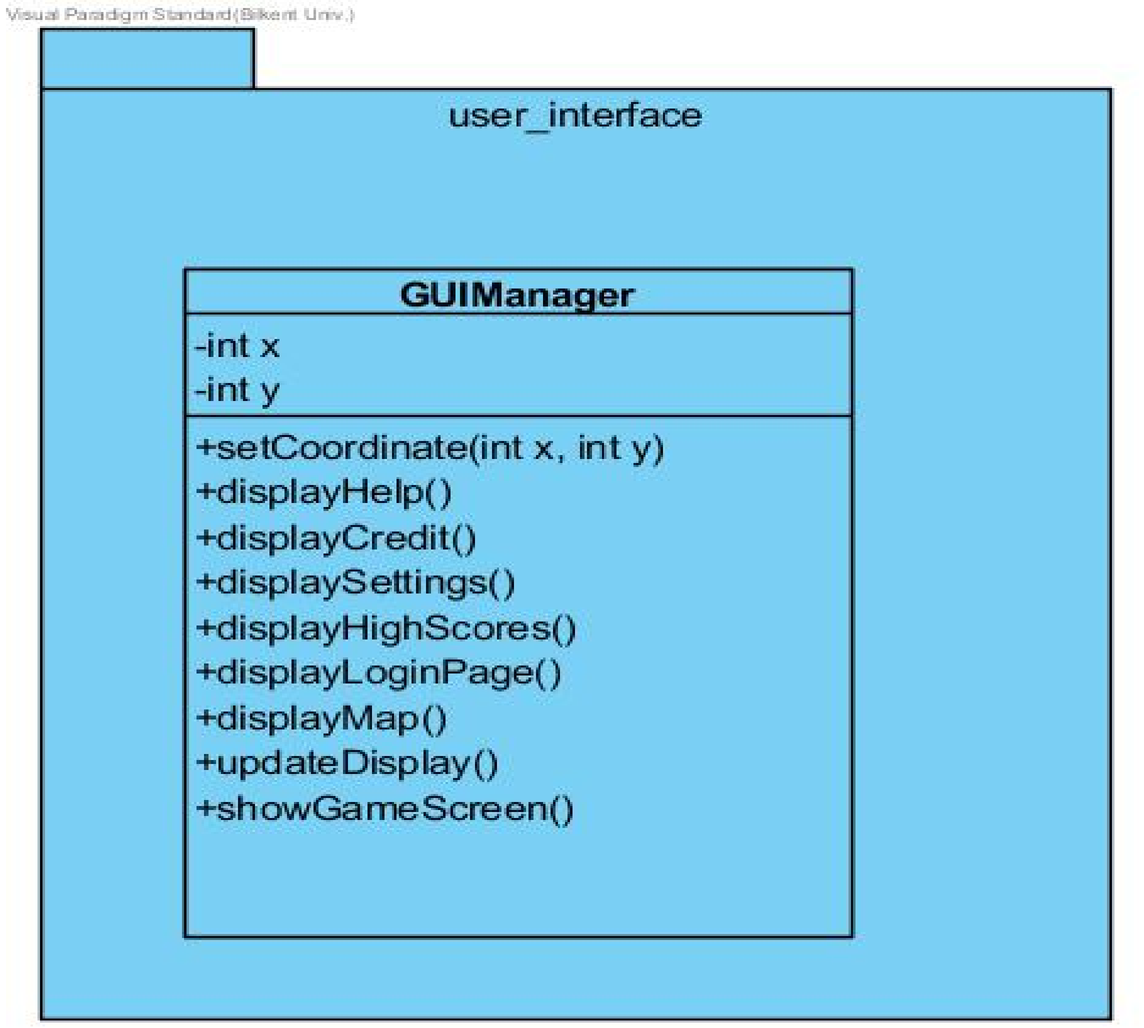


Input manager component is in the second layer of our three-tier architecture. The interaction manager component handles the hardware inputs. It gets inputs from mouse and keyboard. Then it provides the data that includes these inputs to user interface component.

3.4) Interaction

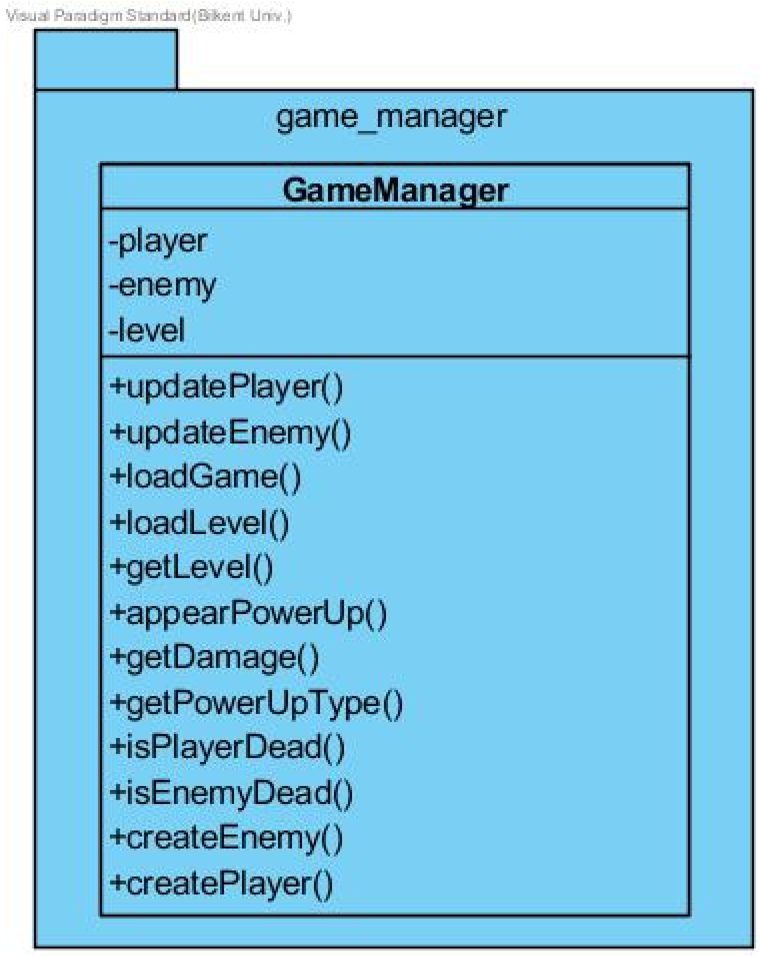
Interactions component is in the second layer of the three-tier architecture. The data about interactions that player has during the game play is in the interaction component. It interacts with game manager and character component. The interaction component provides the data about levels that is passed to character component and takes data about the current data from this component. The interaction component also provides the data about game to game manager component.

3.5) User Interface



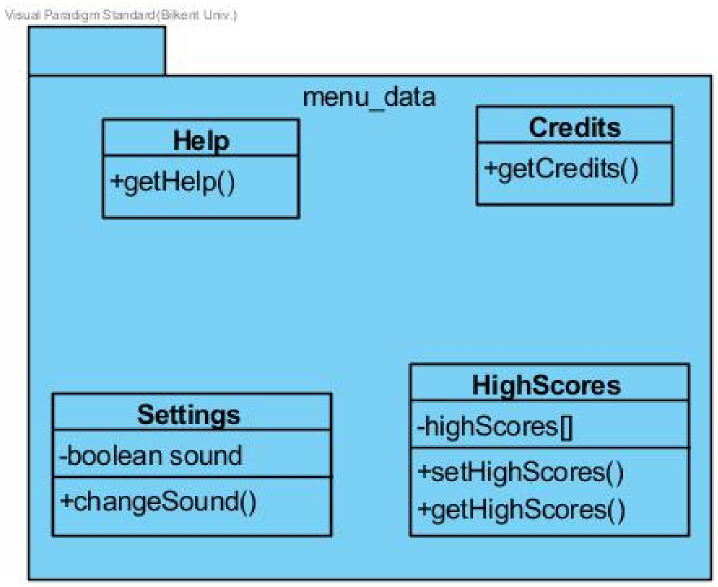
The user interface component is in the first layer of our three-tier architecture. This component takes data from all other components on condition that direct and indirect. Input manager, game manager and menu data components has the direct interaction with user interface component. The input manager provides hardware inputs to the user interface component. The menu data and game manager component also provides the data to user interface component. The menu data shares the menu interaction of the player to user interface. The game manager component provides the game data to the user interface component.

3.6) Game Manager



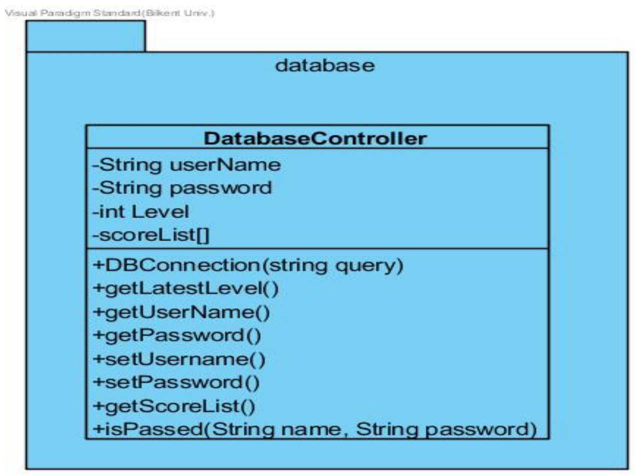
The game manager is in the second layer of the three-tier architecture. It handles all the game play features. The game manager interacts with interaction and user interface components. It takes the game data from interaction component and sends the data of the game to user interface component.

3.7) Menu Data



Menu Data package will place in the second layer of our three tier architecture. This system consists of Help, Credits, Setting and HighScore classes. Menu Data component works connected with Database system to get High Scores' of player. Also, reflecting changes of settings and transition between screens, Menu Data provides information for different menu screens to user interface component.

3.8) Database



Database component takes place in the third layer of our three-tier architecture. This part contains the databaseController class. Basically this system established the connection between database and our game at the bottom layer. User’s latest level, high scores, account information and user’s preferences like character type transferred via this system. Also, changes on data reflect on the database within this package.

4)Low-Level Design

4.1) Object Design Trade-Offs

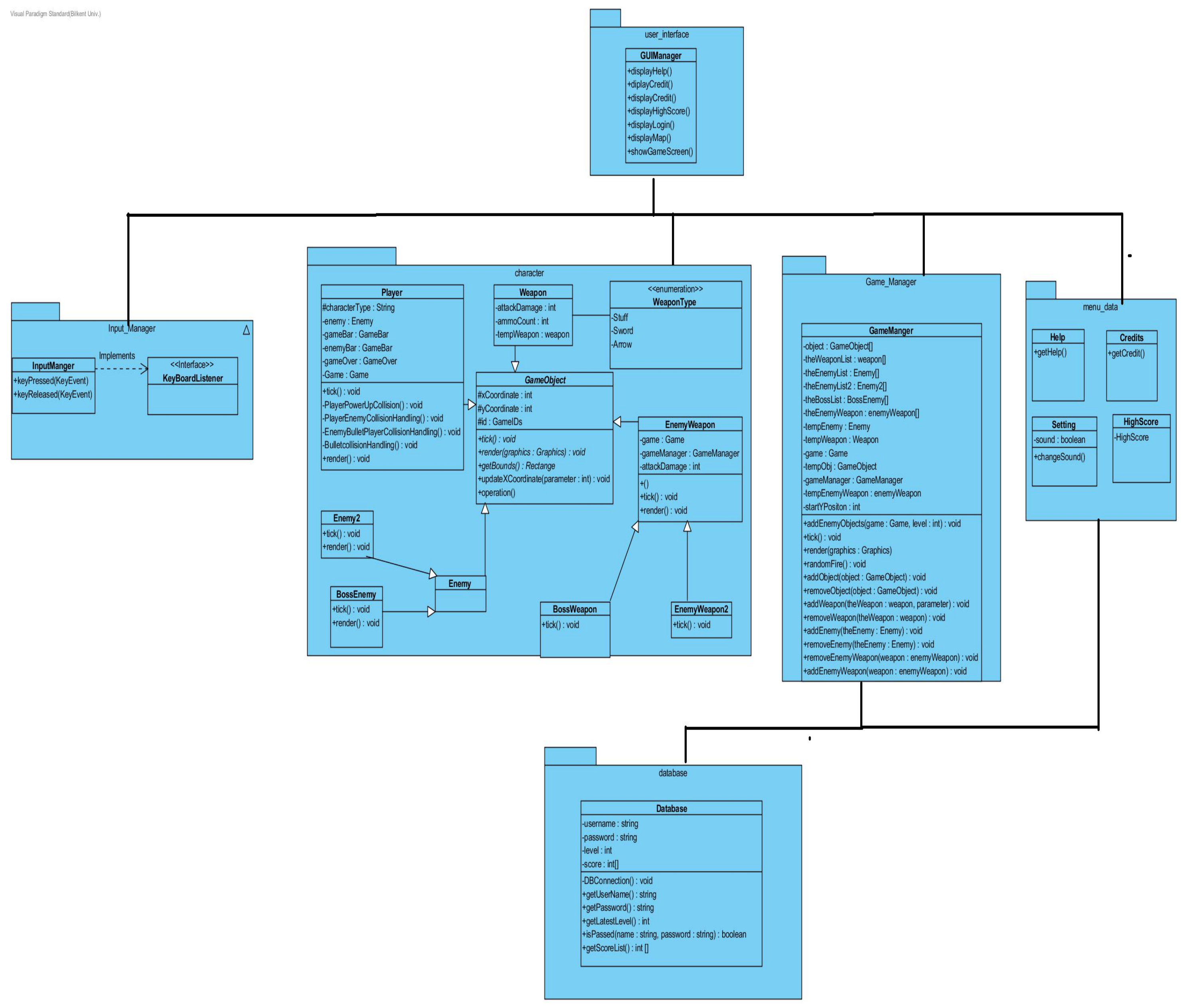
4.1.1)Functional Decomposition vs code length

In our design report, we have aimed at dividing the code into individual modules. By doing so, maintenance and updation of the code can be done in a less strenuous manner and in a more efficient way, but at the cost of increased volume of lines of code.

4.1.2)Performance vs complexity

We have implemented the project by using a three-tier architecture. By doing so we have separated the game’s Graphical User Interface from the Business logic layer of the game, in this case, the game manager which control running of the game. Though the adoption of this architecture the complexity of the game design would decrease which would ultimately ease the implementation of the game, as a complex design results in a complex and hard-to-do implementation.

4.2)Final Object Design



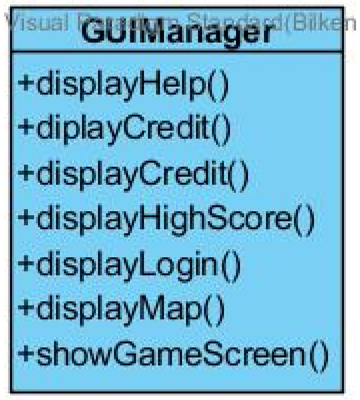
4.3)Packages

As mentioned previously ,the implentation of this project will be realize though the 3-tier architectural style. In view layer of the architecture, we have user_interface package. This package controls the interaction between program and user. In the second layer, we have 5 packages. Input_manager package is for transferring inputs to the control of the software. Character package is for organising user’s data. Interaction package is for controlling interaction between objects in the game. Game_manager is the central package of the game in other words it is basically for whole game process. Menu_data package is for providing menus’ data to the GUIManager package. Finally the Database package provides connection between software and database.

4.4) Class Descriptions

4.4.1) User Interface Class

4.4.1.2) GUI Manager



Operations :

Public void displayHelp() : Displays Help Screen.

Public void displayCredit() : Displays Credit Screen.

Public void displaySettings() : Displays Settings Screen.

Public void displayHighscores() : Displays High Scores Screen.

Public void displayLoginPage(): Display Login page.

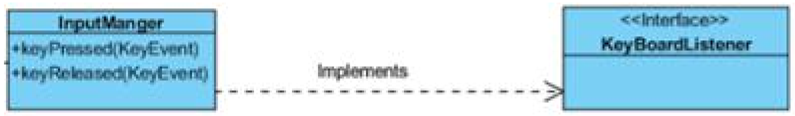
Public void displayMap():Display Map Screen.

Public void *updateDisplay*(): Update the current view.

Public void *showGameScreen(int level)*: Displays the game screen with chosen level.

4.4.2) Input Manager Class

4.4.2.1) Input Manager



This class is for taking input from the keyboard and it implements the KeyBoardListener interface.

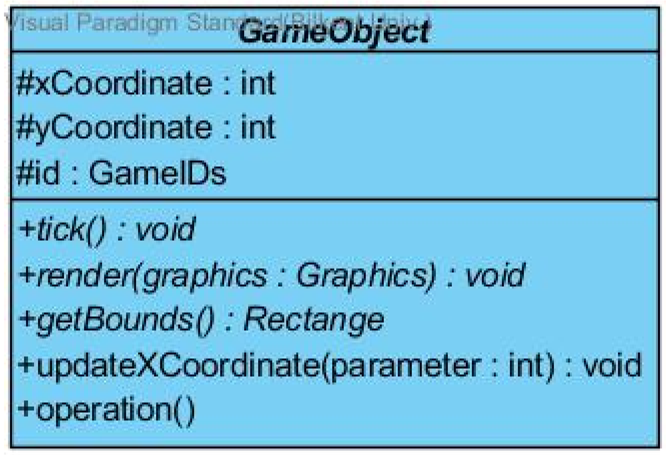
Operations :

Public boolean *keyPressed(KeyEvent)*: Checks the key if it is pressed or not.

Public boolean *keyReleased(KeyEvent)*: Checks the key if it is released or not.

4.4.3)Character Classes

4.4.3.1) GameObject(Abstract class)


Visual Paradigm Standard(Bilkent Univ.)
GameObject
#xCoordinate : int
#yCoordinate : int
#id : GameIDs
+tick() : void
+render(graphics : Graphics) : void
+getBounds() : Rectange
+updateXCoordinate(parameter : int) : void
+operation()

Attributes:

Protected int*x Coordinate*: Object which indicates x coordinate

Protected int*y Coordinate*: Object which indicates y coordinate.

Protected GameIDs *GameIDs*: Game ID object to get enumeration type.

Operations :

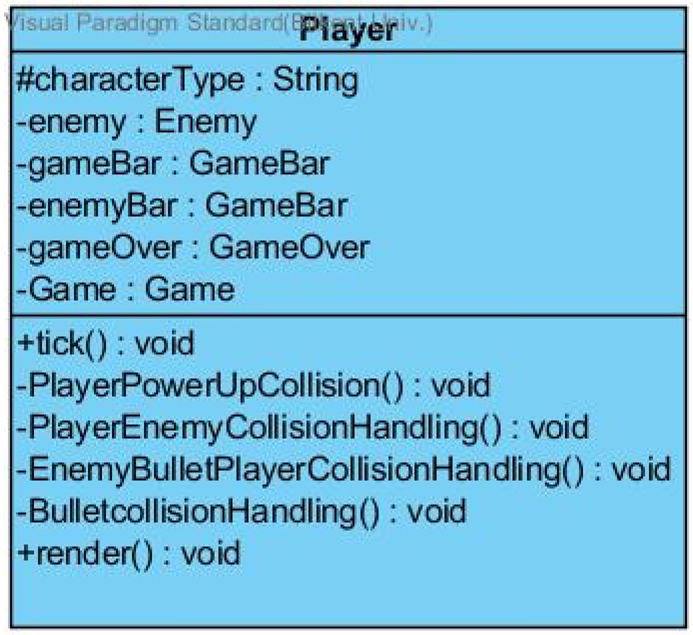
Public void *tick()*: Updates all the game objects phyics

Public void *render*: Renders all the graphics objects of the game

Public Rectangle *getBounds*: Getting the bounds of the rectangle which these rectangles are either player, enemy or bullet.

Public void *updateXCoordinate(int parameter)*: Updates the x coordinate of the enumerated type.

4.4.3.2) Player


Player
#characterType : String
-enemy : Enemy
-gameBar : GameBar
-enemyBar : GameBar
-gameOver : GameOver
-Game : Game
+tick() : void
-PlayerPowerUpCollision() : void
-PlayerEnemyCollisionHandling() : void
-EnemyBulletPlayerCollisionHandling() : void
-BulletcollisionHandling() : void
+render() : void

This class handles character preferences of user during the game.

Attributes:

Private Enemy *enemy*: Enemy object.

Private GameBar *gameBar*: Game bar object.

Private GameBar *enemyBar*: Enemy bar object.

Private GameOver *gameOver*: Game over object.

Private Game*game*: Game object.

Operations :

Public void *tick()*: The following method updates all the physical dynamics of the player object

Private void *playerPowerUpCollision()*: This method determines the collision between player and power-up.

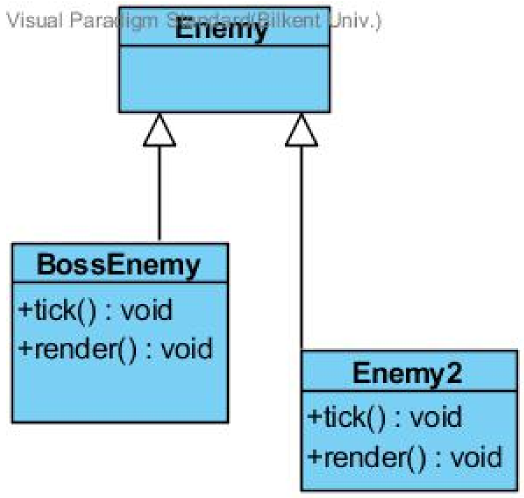
Privatevoid *playerEnemyCollisionHandling()*: This method determines the collision between player and enemy and decrease the health of the player if it is collide.

Private void *bulletCollisionHandling()*: This method determines the collision between player bullet and enemy and decrease the health of the enemy if it is collide.

Private void *enemyBulletPlayerCollisionHandling()*: This method determines the collision between player and enemy bullet and decrease the health of the player if it is collide.

Public void *render()*: This method renders the graphics of the player object.

4.4.3.3) Enemy



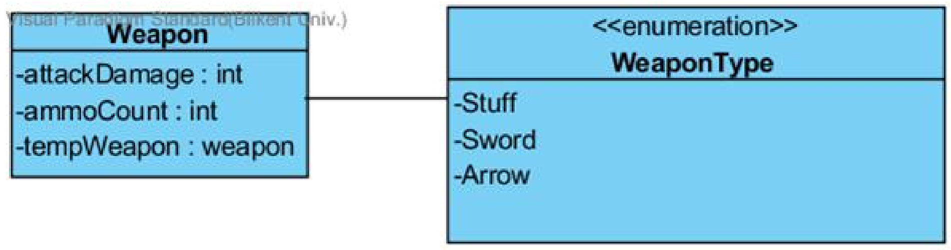
This class is for the creating the enemies. There are two types of enemies, which is the child class of the Enemy class, BossEnemy and Enemy2. These child classes are similar.

Operations :

Public void *tick()*: The following method updates all the physical dynamics of the enemy object

Public void render*()*: This method renders the graphics of the Enemy objects

4.4.3.4) Weapon



This class is the child class of GameObject class. It has enumeration which holds Stuff, Sword and Arrow for the Weapon class.

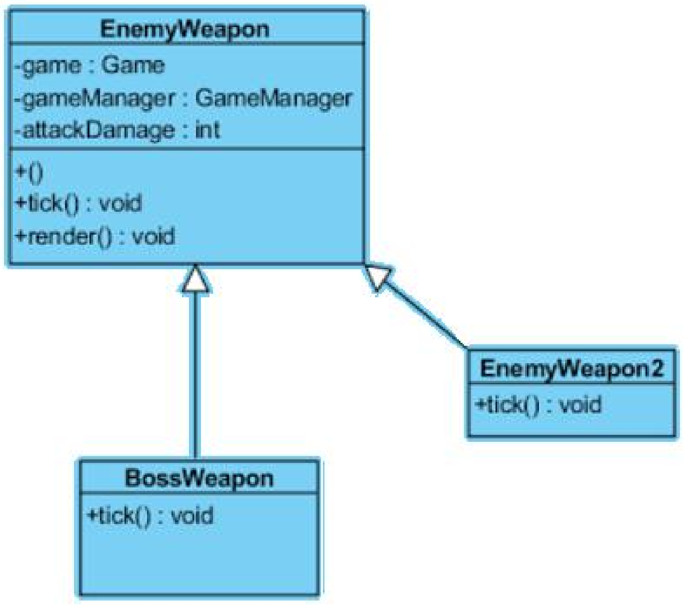
Attributes:

Private int *attackDamage*: This attributes holds the damage as an integer.

Private int *ammoCount*: Holds ammo count as an integer .

Private weapon *tempWeapon*:Temp object which holds weapon object.

4.4.3.5) EnemyWeapons



The EnemyWeapon class is for the creating the enemies weapons. It has two children which are BossWeapon and EnemyWeapon2. These child classes are similar with each other.

Attributes:

Private Game *game*: The game object.

Private GameManager *gameManager*: The game manager object.

Private int *attackDamage:* The object to count the attack damage.

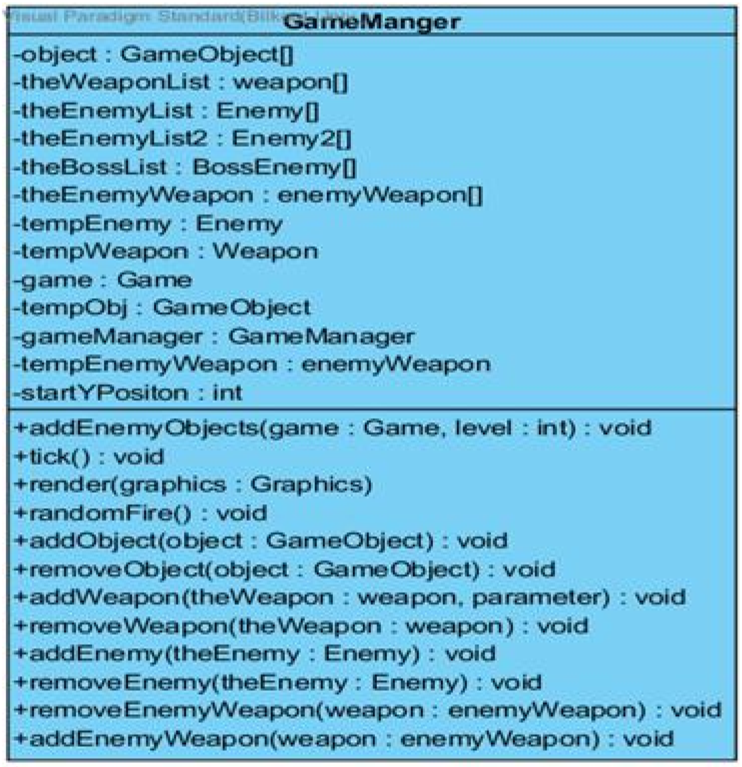
Operations:

Public void *tick()*: method updates all the physical dynamics of the enemy bullets.

Public void render*()*: Renders the graphics of the enemy weapon.

4.4.4) Game Manager Class

4.4.4.1) Game Manager


GameManger
-object : GameObject[]
-theWeaponList : weapon[]
-theEnemyList : Enemy[]
-theEnemyList2 : Enemy2[]
-theBossList : BossEnemy[]
-theEnemyWeapon : enemyWeapon[]
-tempEnemy : Enemy
-tempWeapon : Weapon
-game : Game
-tempObj : GameObject
-gameManager : GameManager
-tempEnemyWeapon : enemyWeapon
-startYPositon : int
+addEnemyObjects(game : Game, level : int) : void
+tick() : void
+render(graphics : Graphics)
+randomFire() : void
+addObject(object : GameObject) : void
+removeObject(object : GameObject) : void
+addWeapon(theWeapon : weapon, parameter) : void
+removeWeapon(theWeapon : weapon) : void
+addEnemy(theEnemy : Enemy) : void
+removeEnemy(theEnemy : Enemy) : void
+removeEnemyWeapon(weapon : enemyWeapon) : void
+addEnemyWeapon(weapon : enemyWeapon) : void

The GameManager class is for handling all the game play features.

Attributes:

Private GameObject[]*object*: The list of game objects.

Private weapon[]*theWeaponList*: The list of weapons.

Private Enemy[]*theEnemyList:* The list of enemies for level 1.

Private Enemy2[]*theEnemyList2*: The list of enemies for level 2.

Private BossEnemy[]*theBossList*: The list of boss enemy list for level 3.

Private enemyWeapon[]*theEnemyWeapon*: The list of the enemy weapon object for bullets.

private Enemy*tempEnemy*: Temp object for enemy.

Private Weapon*tempWeapon*: Temp object for weapon.

Private Game*game*: The game object.

Private GameObject*tempObj*: A temp object for game object.

Private GameManager*gameManager*: The game manager object.

Private enemyWeapon*tempEnemyWeapon*: The temp enemy weapon object.

Private int *startYposition*: The starting point.

Operations:

Public void *tick()*: Call all the tick of the classes that used it in order to update the game physical dynamics

Public void addEnemyObjects*(Game game, int level)*: This method adds enemy object for the specific level.

Public void render*(Graphics graphics)*: Renders the graphics objects.

Public void randomFire*()*: This method is for make the enemy randomly fire to the player.

Public void addObject*(GameObject object)*: Adds object into the game play.

Public void removeObject*(GameObject object)*: Removes object from the game play.

Public void addWeapon*(weapon theWeapon, int parameter)*: Adds the weapon into the game play.

Public void removeWeapon*(weapon theWeapon)*: Removes the weapon from the game play.

Public void addEnemy*(Enemy theEnemy)*: Adds enemy into the game play.

Public void removeEnemy*(Enemy theEnemy)*: Removes enemy from the game play after its health finishes.

Public void removeEnemyWeapon*(enemyWeapon weapon)*: Removes the enemy weapon forn the game play.

Public void addEnemyWeapon*(enemyWeapon weapon)*: Adds the enemy weapon into the game play.

4.4.5) Menu Data Classes

4.4.5.1) Help Class

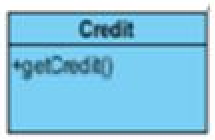
Help
+getHelp()

Help class allows to get the game information that we will write.

Operations :

Public String *getHelp*(): This method is for getting help information.

4.4.5.2) Credits Class

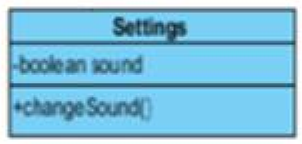


Credit class allows to get the information about developers that we will write.

Operations :

Public String *getCredit*(): This method is for getting credit informations.

4.4.5.3) Settings Class



Settings class allows to make changes the sound settings.

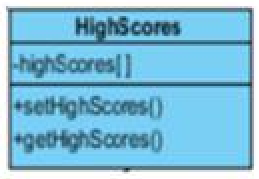
Attributes:

Private boolean *sound*: Holds a boolean object that shows the sound open or close.

Operations :

Public void *changeSound(sound)*: This method is for close or open the sound.

4.4.5.4) High Score Class



High Scores class deals with the setting and getting high score values.

Attributes:

Private int *highScores*[]: The array that holds the high scores.

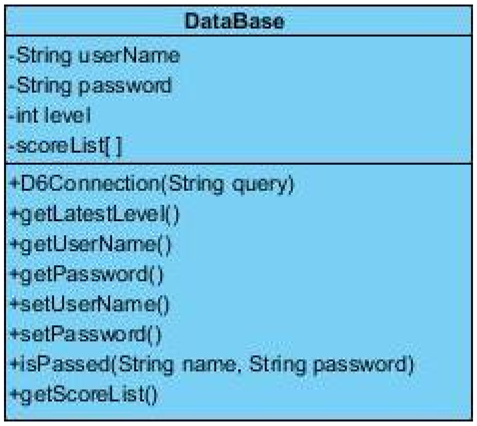
Operations :

Public void *setHighScores*(): This method is for setting the high scores the player has during the game play.

Public int *getHighScores*(): This method is for getting high score informations.

4.4.6) DataBase Class

4.4.6.1) DataBase



This class handles the connection part between game and database such as retrieving and sending data.

Attributes:

Private String *username*: Holds the username.

Private String *password*: Holds the user's password.

Private int *level*: Lates level played.

Private int [] *scores*: List of high scores.

Operations :

Private void *DBConnection(String query)*: Ensure the connection between MySQL database and our program.

Public int *getLatestLevel*(): Retrieve the last level player played.

Public String *getUsername*(): Return username.

Public String *getPassword*(): Return password.

Public String *setPassword*(): Change the current account's password.

Public String *setUsername*():Change the current account's username.

Public int[] *getScorelist*(): Return the list of high scores of the current player.

Public String [] *getLatestCharacter(Account user)*: List of preferences of character(type, damage…) lastly modified by user.

Public Boolean *levelAvailable(int level)*: Check the availability of the level.

Public Boolean *isPassed(String name, String password)*: If given information is matched, it gives the permission.