CS 319 - Object-Oriented Software Engineering

Design Report

Nightmare Dungeon

Group 3-J

Berk Mandıracıoğlu

Mehmet Oğuz Göçmen

Hüseyin Emre Başar

Hakan Sarp Aydemir

Table Of Contents

1 Introduction

1.1 Purpose of the system

The system of Nightmare Dungeon's purpose is to make users have fun from the game that presents a different experience in every run because of the random elements of the rogue-like genre such as different item and minion spawns. The gameplay and the controls are inspired by BoE so they are easy to learn so the player can enjoy the game with little effort invested beforehand. Although the game is easy to learn, the game can get challenging so the player can find excitement about the game.

1.2. Design Goals

Design step is one of the most important step in project development. In the procedure of designing, we are required to define design goals. Following descriptions include important design goals.

1.2.1. Criteria

End User Criteria

- **Usability:** The game will be user-friendly in its design. A new player will be able to start the game and jump into the game directly with a simple UI that isn't clustered with confusing buttons and that guides the user around.

- **Performance:** The controls are going to be responsive to keep up with the pace of the game. The game will run smoothly in order to provide a stable fps and enjoyable gameplay.

Maintenance Criteria

- **Reliability:** The system should always give the user the promised service so we will make the game such that it will not crash or have game breaking bugs. To achieve this kind of reliability we will frequently test the system and debug the code.

- **Good Documentation:** We will have good documentation of our system so that it is easy to work on, traceable and maintainable. We will do this by saving our drafts and work progress as well as the final product so that we can follow through the development process in its every step as we need it.

- **Extendibility:** The system will be easy to extend so that new features and functionalities will be easy to add. We will accomplish this by following object oriented design and its methods such as abstraction from start to end. By doing so, we will be able to add or modify a class without corrupting the functionalities of other classes or disturbing other elements of the design.

1.3. Definitions, acronyms and abbreviations

BoE [1]: BoE is an acronym used in the Binding of Isaac community to refer to Binding of Isaac.

Fps [2]: Fps is an acronym for frame per second which is the count of frames that the screen displays in a second.

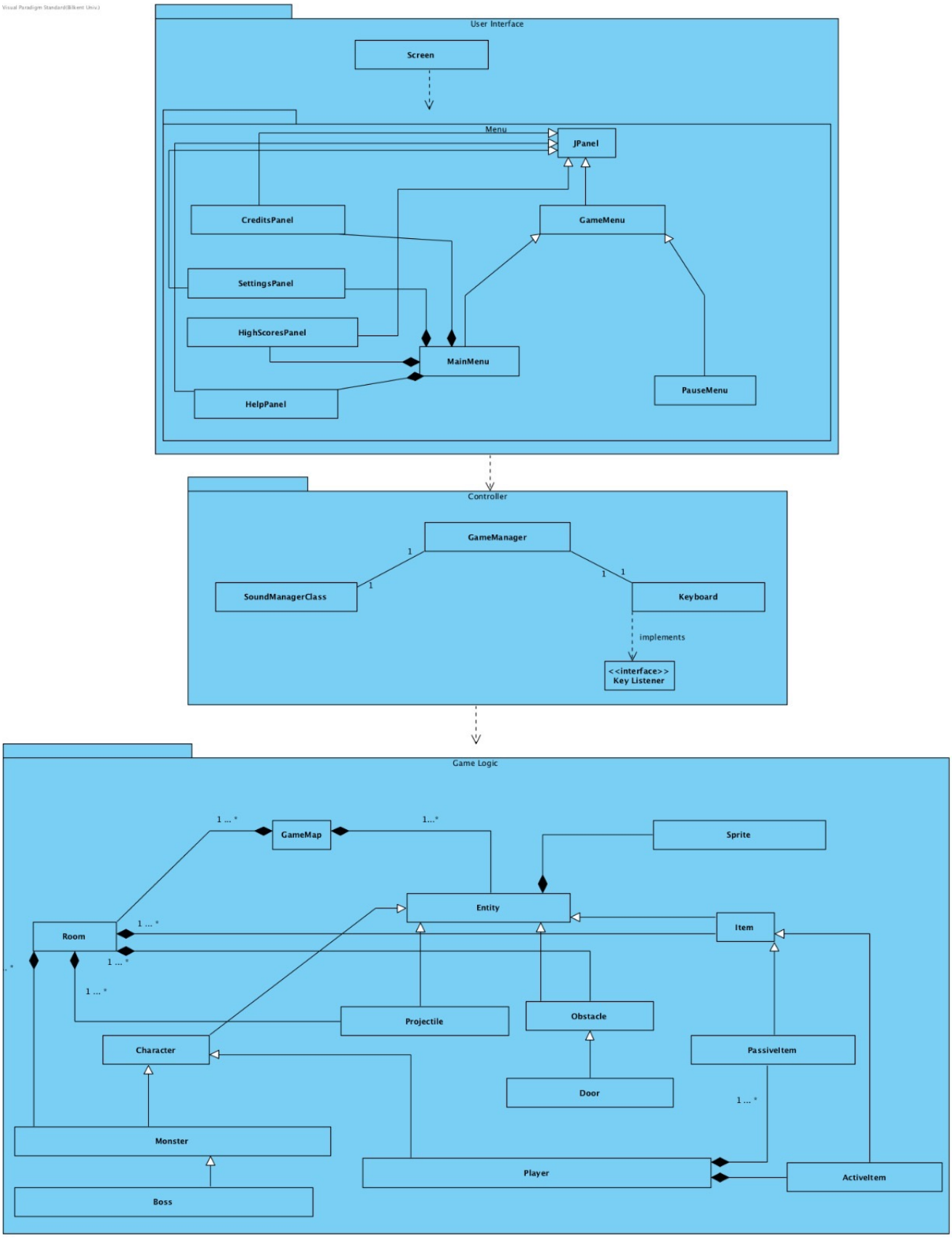
UI [3]: UI is an acronym for user interface which is a design that allows the user and the system to interact.

Rogue-like [4]: Roguelike is a term used to describe a subgenre of role-playing video games that are characterized by a dungeon crawl through procedurally generated game levels, turn-based gameplay, tile-based graphics, and permanent death of the player-character.

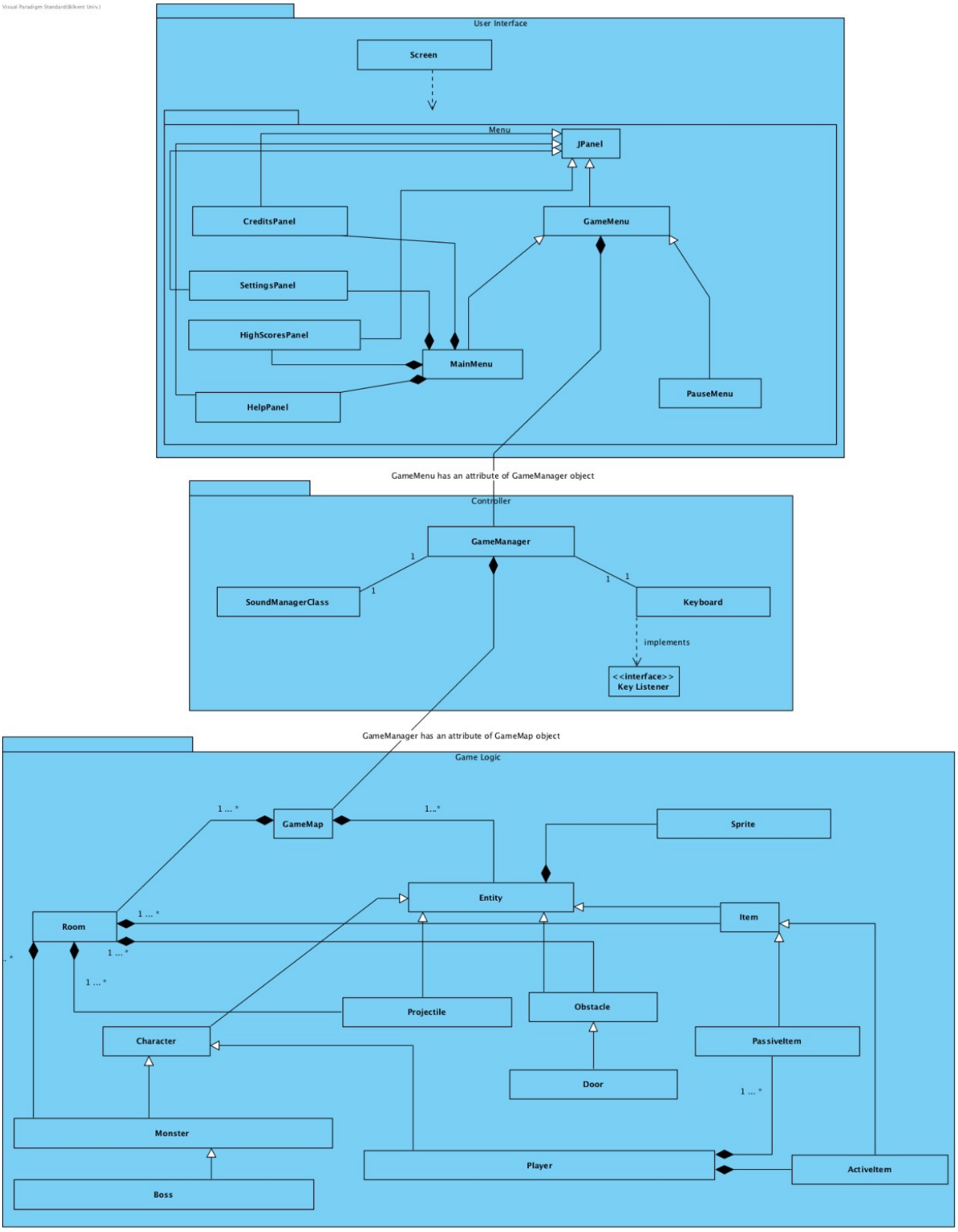
2. Software Architecture

2.1. Subsystem Decomposition

As for the architecture, we chose to go with the three-tier architecture for the design of the system since most games are heavily dependent on game logic, user interface and the interaction between them so we thought that these three main elements that make up most of the games can be categorized into three layers. These three layers will provide MVC design by acting as view (User Interface), controller (Game Managers) and model (Game Logic). View is the top layer and it will act as an interaction between the user and the system and it will depend on controller. Controller is the middle layer and it will act as a connection between view and the controller. Model is the bottom layer and it will handle the data flow of the game. By following MVC we are aiming to create a system with high coherence and low coupling. We used opaque layering as a layering method.



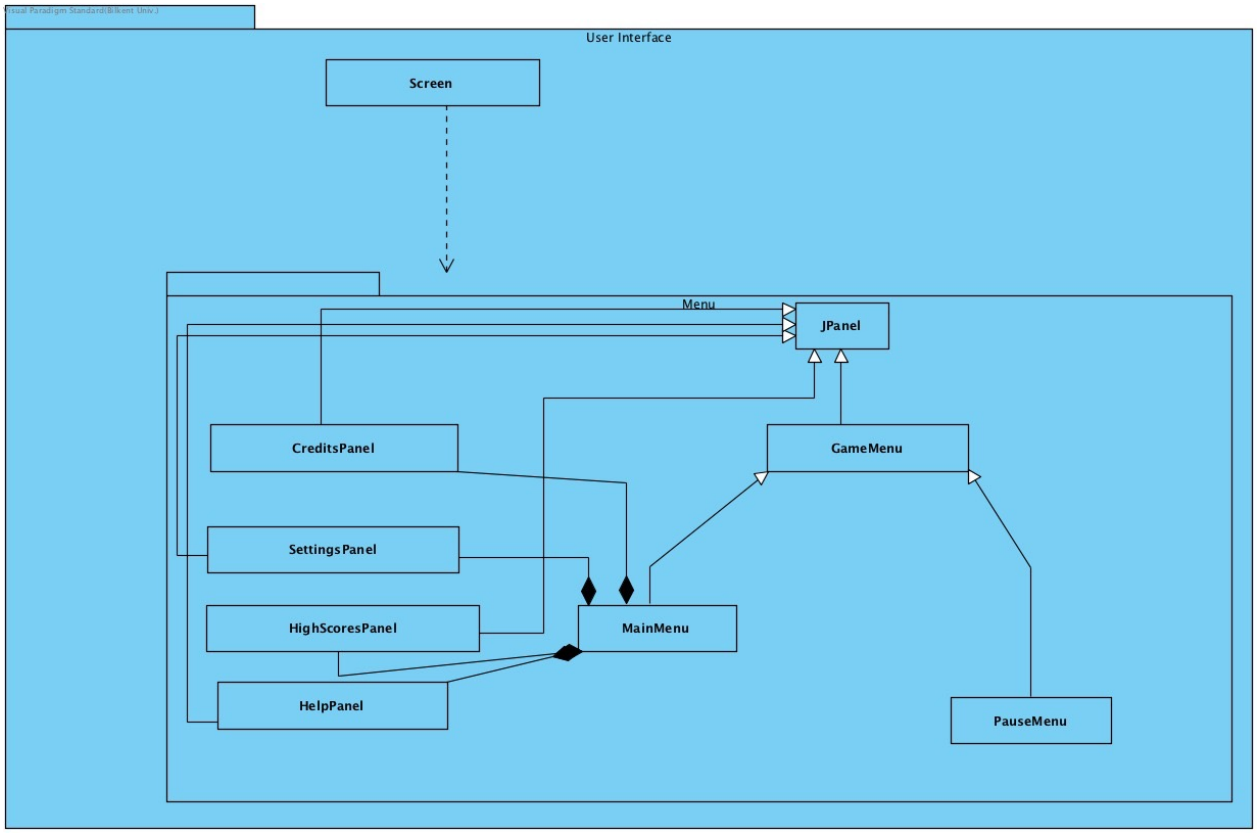
Basic Subsystem Decomposition



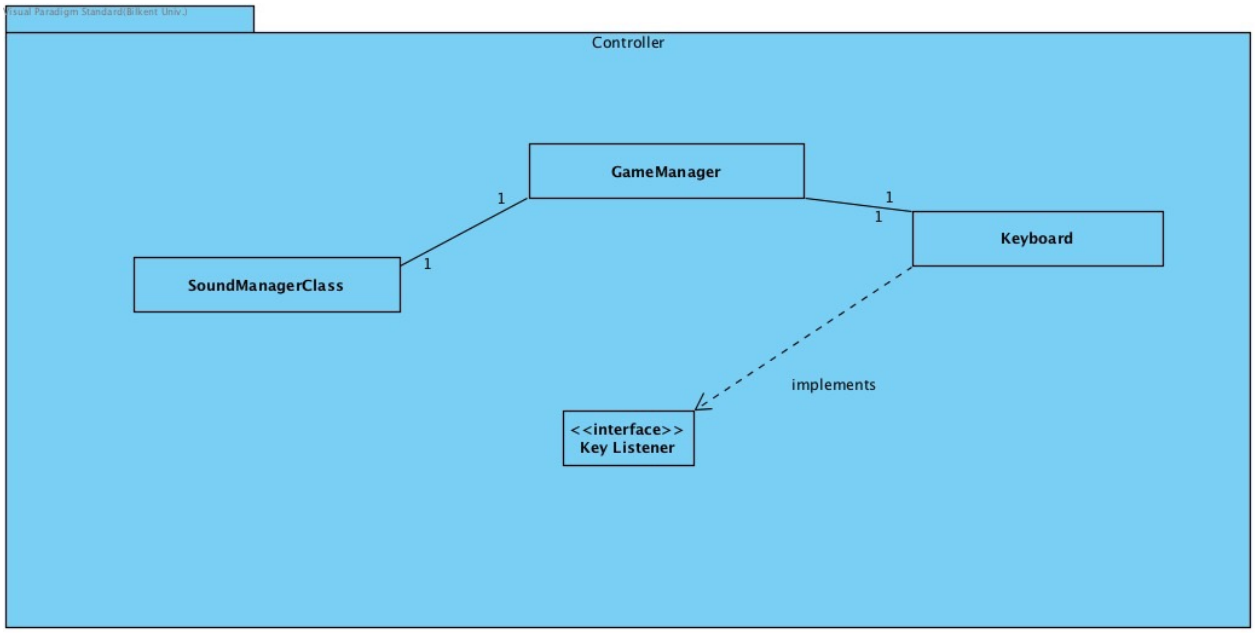
Detailed Subsystem Design

High Level View of Subsystem Decomposition

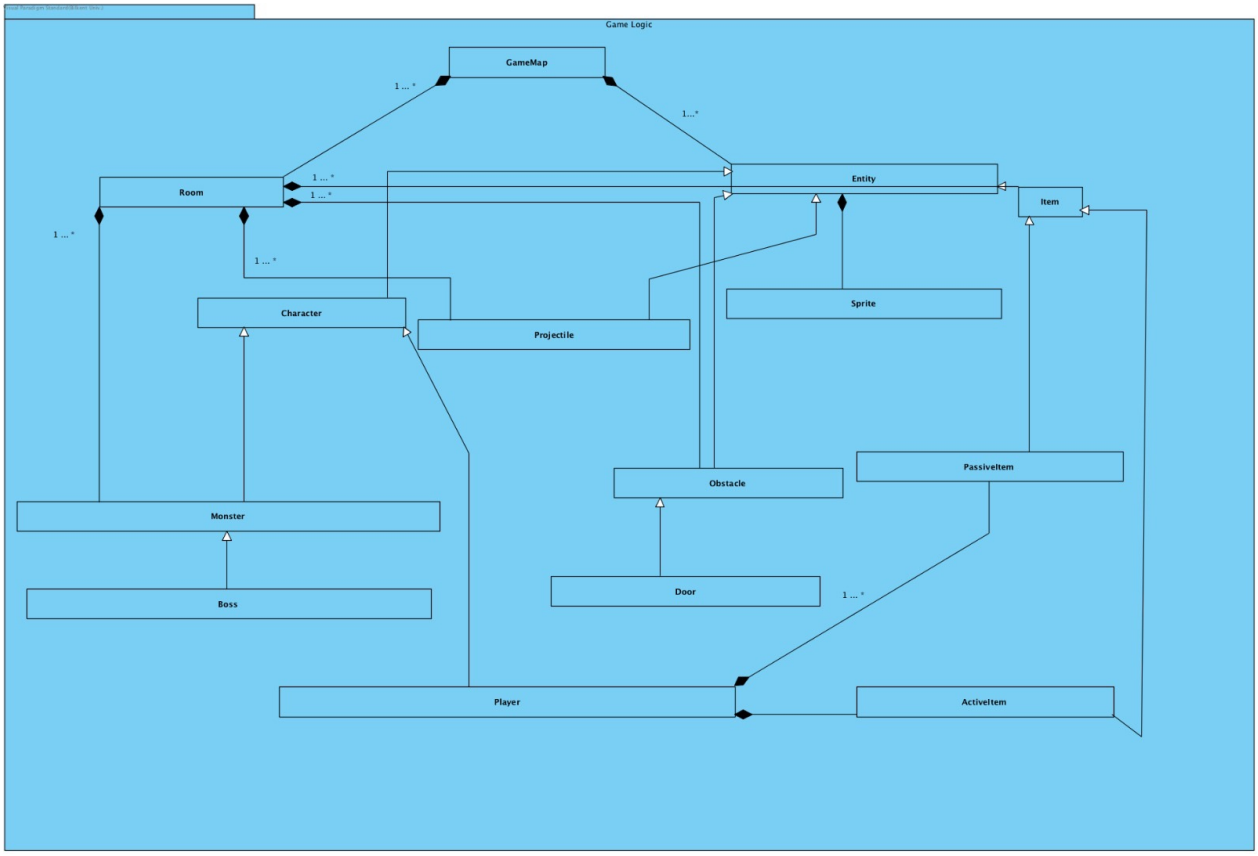
User Interface Subsystem



Controller Subsystem



Game Logic Subsystem



2.4. Hardware / Software Mapping

As software configuration, we will implement the game in Java.

As hardware configuration, the user will need a keyboard for controlling the character and writing his/her name on the high scores list and a computer screen for the user to interact with the system. A PC with Microsoft Windows and Java compiler will be able to run the game so the system requirements are minimal. We will use local files such as .txt files to store game data such as high scores list so

the user won’t need an internet connection to play the game or submit his/her high score to the top 10 list.

2.5. Persistent Data Management

We will store our game’s data locally so the user won’t need internet and we won’t need to build a database system. We will store the high scores in a single file and get the data from there every time player opens the game. We will store the character models in sprite-sheets as sprites so the game will get the pixel data from the sprite-sheet file when needed. We will also have sound files such as .mp3’s for the background music and the sound effects.

2.6. Access Control and Security

Nightmare Dungeon doesn’t have any interaction with internet connection and it doesn’t have a database that holds user information. The only user information the game is going to need is a name when one of the top 10 high scores is beaten to replace the position whose score was beaten by the current player. At this point the player can use any alias he/she wants. Since this is the case, players won’t need an account so the system doesn’t have any access control or security issues.

2.7. Boundary Conditions

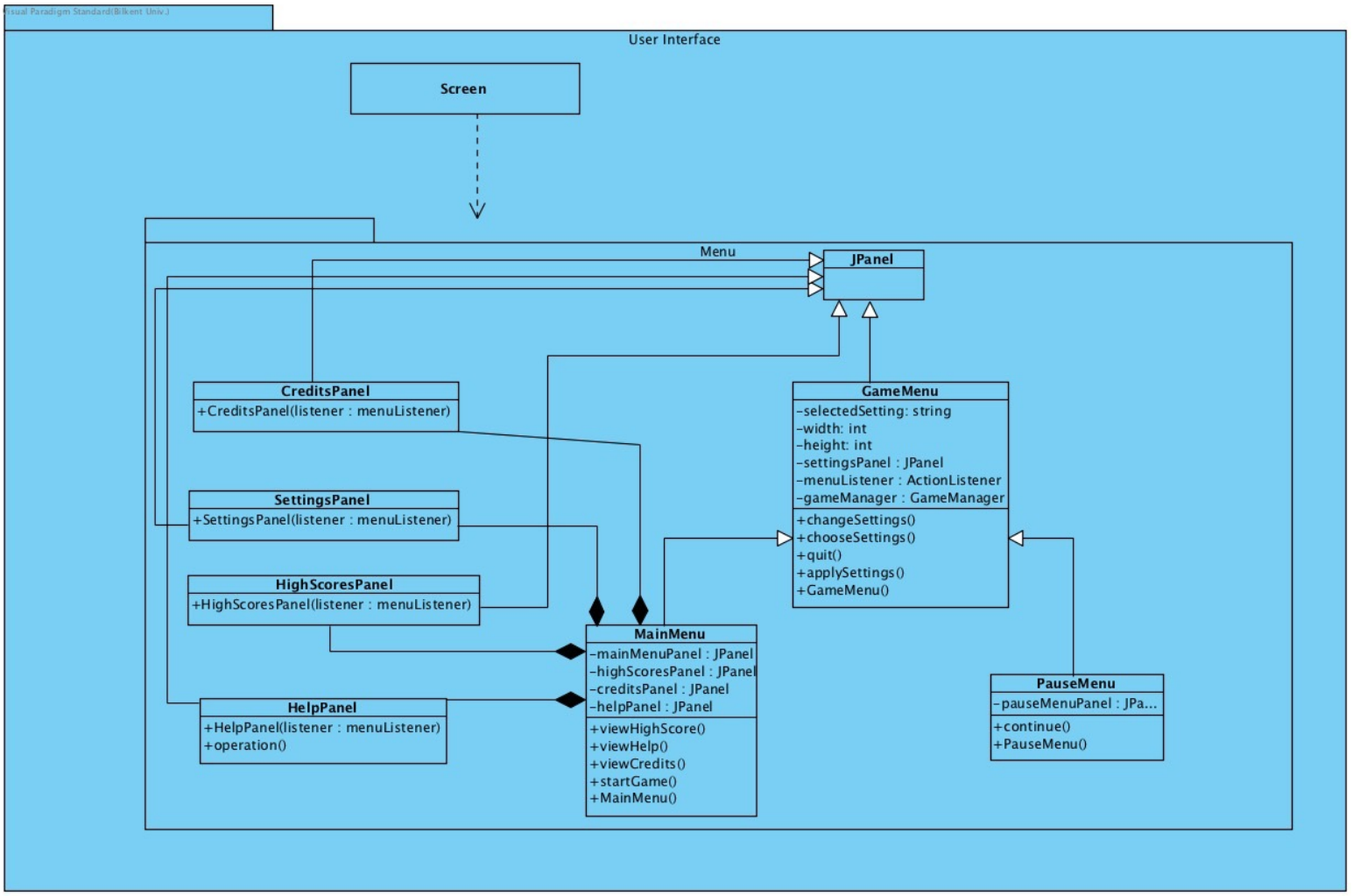
The maps will have boundaries on the edges, the implementation will include cases where the player tries to go beyond these boundaries and the game will not allow going beyond the boundaries. We will still catch exceptions about boundaries even if we have checks controlling the boundaries of the map to avoid program crashes. Another boundary case is the situation where all of the player's lives are gone. When this happens, the game will be over and it will start again. Another boundary case is when the player manages to kill the last boss. When this happens, the game will be over and the players score will be displayed, if the player beats one of the top 10 scores previously saved, then the game will request a name from the user to save it to the top 10 board along with the score of the player.

3. Subsystem Services

3.2. User Interface Subsystem

The interaction between the user and the system is provided by the User Interface subsystem. This interaction is done with the use of graphical components. The User Interface subsystem consists of 4 classes which are GameMenu,

MainMenu, PauseMenu and Screen.



3.2.1. Screen Class

Screen class handles graphics objects such as JFrames and JPanels. Every component related to graphics will be put over the JFrame created by the Screen class. The main JPanel of the game called gamePanel will be created and as data changes in the Game Logic section, the screen class will render() to display these changes to the user.

3.2.2. JPanel Class

The JPanel class will act as a parent to all the other classes in the Menu package.

3.2.3. GameMenu Class

The GameMenu class is a child of the JPanel class and the parent of MainMenu and PauseMenu classes. GameMenu class has the common funclionalities of the MainMenu and PauseMenu classes such as quit(), changeSettings(), selectedSetting, etc.

3.2.4. MainMenu Class

MainMenu class is the first class to be created when the game is opened first with the use of the constructor MainMenu(). The MainMenu class presents 4 choices unique to the MainMenu class when the user is in the main menu. These choices are: viewHelp(), viewHighScore(), viewCredits() and startGame(). startGame() method initiates the game. MainMenu class also has changeSettings(), chooseSettings(), quit() and applySetting() methods that it shares with the PauseMenu class. MainMenu class also holds JPanels of the options the user can choose while he/she is in the main menu.

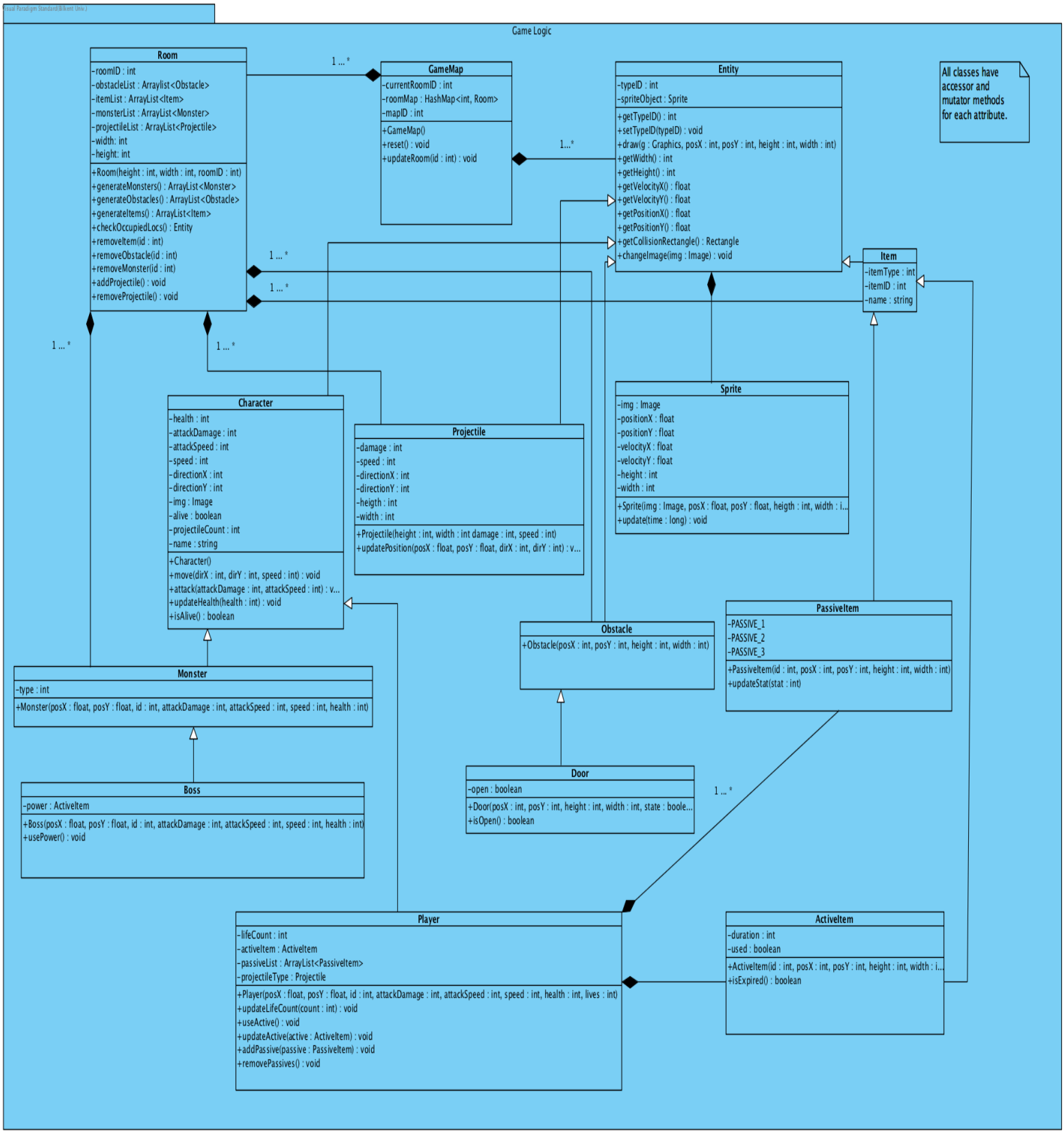
3.2.5. PauseMenu Class

PauseMenu class has the common methods mentioned in the MainMenu class explanation above. It also has a unique continue() method that allows the user to exit from the pause menu and continue playing the game.

3.4. Game Logic Subsystem Interface

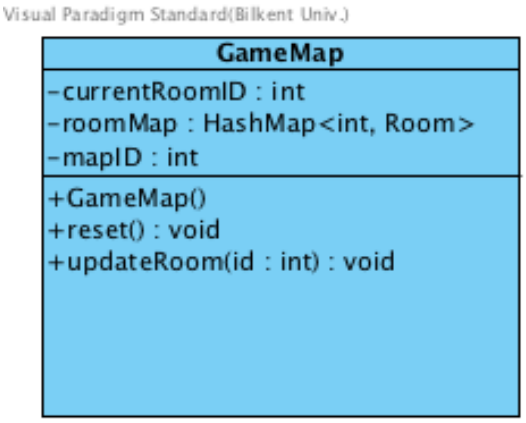
Game Logic subsystem will be the bottom layer in three-tier system which will contain the key components of the product. Key class in this subsystem is GameObject class which is an abstract class for nearly every class in following subsystem.

GameObject class is the keystone for 10 classes in the subsystem. Moreover, there is Room class and GameMap class for arranging each room component inside the game map. There are also compositions between GameMap – Room, GameMap – GameObject and GameObject – Sprite classes.


Game Logic
Room
-roomID : int
-obstacleList : Arraylist<Obstacle>
-itemList : ArrayList<Item>
-monsterList : ArrayList<Monster>
-projectileList : ArrayList<Projectile>
-width: int
-height: int
+Room(height : int, width : int, roomID : int)
+generateMonsters() : ArrayList<Monster>
+generateObstacles() : ArrayList<Obstacle>
+generateItems() : ArrayList<Item>
+checkOccupiedLocs() : Entity
+removeItem(id : int)
+removeObstacle(id : int)
+removeMonster(id : int)
+addProjectile() : void
+removeProjectile() : void
1 ... *
GameMap
-currentRoomID : int
-roomMap : HashMap<int, Room>
-mapID : int
+GameMap()
+reset() : void
+updateRoom(id : int) : void
1...*
Entity
-typeID : int
-spriteObject : Sprite
+getTypeID() : int
+setTypeID(typeID) : void
+draw(g : Graphics, posX : int, posY : int, height : int, width : int)
+getWidth() : int
+getHeight() : int
+getVelocityX() : float
+getVelocityY() : float
+getPositionX() : float
+getPositionY() : float
+getCollisionRectangle() : Rectangle
+changeImage(img : Image) : void
All classes have accessor and mutator methods for each attribute.
Item
-itemType : int
-itemID : int
-name : string
Character
-health : int
-attackDamage : int
-attackSpeed : int
-speed : int
-directionX : int
-directionY : int
-img : Image
-alive : boolean
-projectileCount : int
-name : string
+Character()
+move(dirX : int, dirY : int, speed : int) : void
+attack(attackDamage : int, attackSpeed : int) : v...
+updateHealth(health : int) : void
+isAlive() : boolean
Projectile
-damage : int
-speed : int
-directionX : int
-directionY : int
-heigth : int
-width : int
+Projectile(height : int, width : int damage : int, speed : int)
+updatePosition(posX : float, posY : float, dirX : int, dirY : int) : v...
Sprite
-img : Image
-positionX : float
-positionY : float
-velocityX : float
-velocityY : float
-height : int
-width : int
+Sprite(img : Image, posX : float, posY : float, heigth : int, width : i...
+update(time : long) : void
Obstacle
+Obstacle(posX : int, posY : int, height : int, width : int)
PassiveItem
-PASSIVE_1
-PASSIVE_2
-PASSIVE_3
+PassiveItem(id : int, posX : int, posY : int, height : int, width : int)
+updateStat(stat : int)
Monster
-type : int
+Monster(posX : float, posY : float, id : int, attackDamage : int, attackSpeed : int, speed : int, health : int)
Boss
-power : ActiveItem
+Boss(posX : float, posY : float, id : int, attackDamage : int, attackSpeed : int, speed : int, health : int)
+usePower() : void
Door
-open : boolean
+Door(posX : int, posY : int, height : int, width : int, state : boole...
+isOpen() : boolean
Player
-lifeCount : int
-activeItem : ActiveItem
-passiveList : ArrayList<PassiveItem>
-projectileType : Projectile
+Player(posX : float, posY : float, id : int, attackDamage : int, attackSpeed : int, speed : int, health : int, lives : int)
+updateLifeCount(count : int) : void
+useActive() : void
+updateActive(active : ActiveItem) : void
+addPassive(passive : PassiveItem) : void
+removePassives() : void
ActiveItem
-duration : int
-used : boolean
+ActiveItem(id : int, posX : int, posY : int, height : int, width : i...
+isExpired() : boolean

Figure

3.4.1. GameMap Class



→ This class is the top class of the Game Entities subsystem; it contains all Room objects which essentially generate every core component of the game excluding Player object. In other words, GameMap handles every level in the game.

Attributes:

private int currentRoomID: This attribute holds an integer to represent a roomID to determine which room is currently played or used when calling updateRoom method.

private HashMap<int, Room> roomMap: This map hashes Rooms with their id values.

private int MapID: This integer value holds the value of id of the Map.

Constructor:

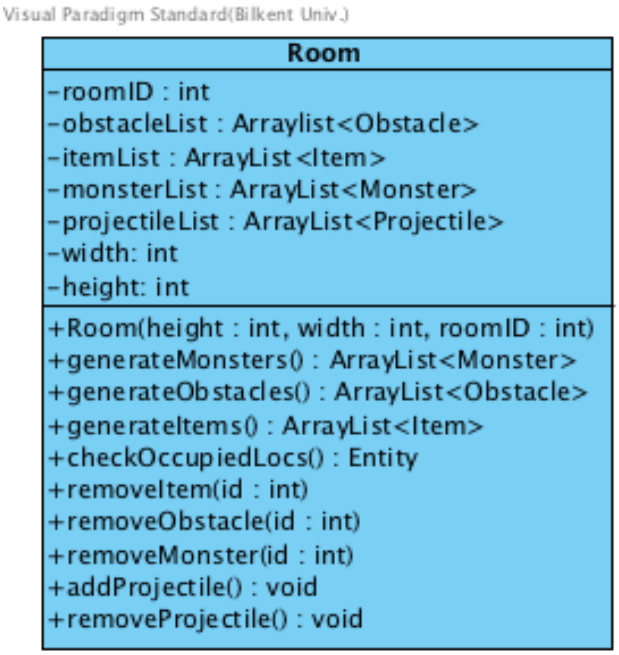
GameMap (): Default constructor with default attributes.

Methods:

public void reset(): This method resets everything in the Map class to the default values.

public void updateRoom (int id): This method changes the current room inside the Map classes with respect to given id.

3.4.2 Room Class



→ Room class contains all the components of the game; all rooms are initialized from the start of the game with Monster, Obstacle and Item objects. This class also handles boundary problems, checks for occupied locations in each room and generate objects accordingly. Each room has a roomID to select the target room accordingly.

Attributes:

private int roomID: This integer value holds the id of the current room.

private ArrayList<Obstacle> obstacleList: This list contains the Obstacles objects inside the Room.

private ArrayList<Item> itemList: This list contains the Items objects inside the Room.

private ArrayList<Monster> monsterList: This list contains the Monsters objects inside the Room.

private ArrayList<Projectile> projectileList: This list contains the Projectiles objects inside the Room.

private int width: This integer value holds the width of the room in screen.

private int height: This integer value holds the height of the room in screen.

Constructor:

Room (int height, int width, int roomID): Initializes the Room object with the given width, height and roomID and gives default values to attributes.

Methods:

public ArrayList<Monster> generateMonsters (): This method spawns the Monster objects inside the room and puts them to monsterList. Every Monster spawns in kind of random locations without any collisions with other objects.

public ArrayList<Obstacle> generateObstacles (): This method generates Obstacle objects inside the room and puts them to obstacleList. Every Obstacle is generated in kind of random locations without any collisions with other objects.

public ArrayList<Item> generateItems (): This method generates Item objects inside the room and puts them to itemList. Every Item is generated in kind of random locations without any collisions with other objects.

public Entity checkOccupiedLocs(posX,posY): This method checks if there is any collision with player and other objects inside the room. Room traverses inside its object lists and checks each of them. If there is a collision it returns the collided Entity objects.

public void removeItem (int id): This method removes the item with the given id from the itemList inside the Room class.

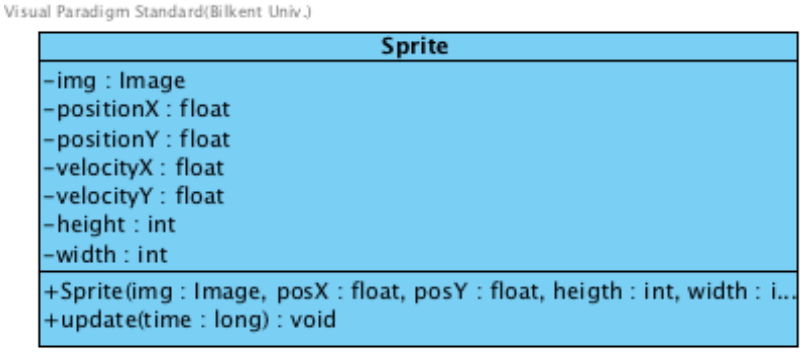
public void removeObstacle (int id): This method removes the obstacle with the given id from the obstacleList inside the Room class.

public void removeMonster (int id): This method removes the monster with the given id from the monsterList inside the Room class.

public void addProjectile (): This method adds projetile to the projectileList inside the Room class.

public void removeProjectile (): This method removes projetile from the projectileList inside the Room class.

3.4.3 Sprite Class



→ Sprite class' purpose is to update the position of Sprite object which contains an Image object inside. It has a method called "update" which manipulates the position of Sprite object in a given time interval by multiplying velocity values with given time interval to produce new positions.

Attributes:

private Image img: Image object for the appearance of Sprite object which will be manipulated over time.

private float positionX: x coordinate of the top left corner of object.

private float positionY: y coordinate of the top left corner of object.

private float velocityX: x coordinate of the velocity of object.

private float velocityY: y coordinate of the velocity of object.

private int height: This integer value holds the height of the object.

private int width: This integer value holds the width of the object.

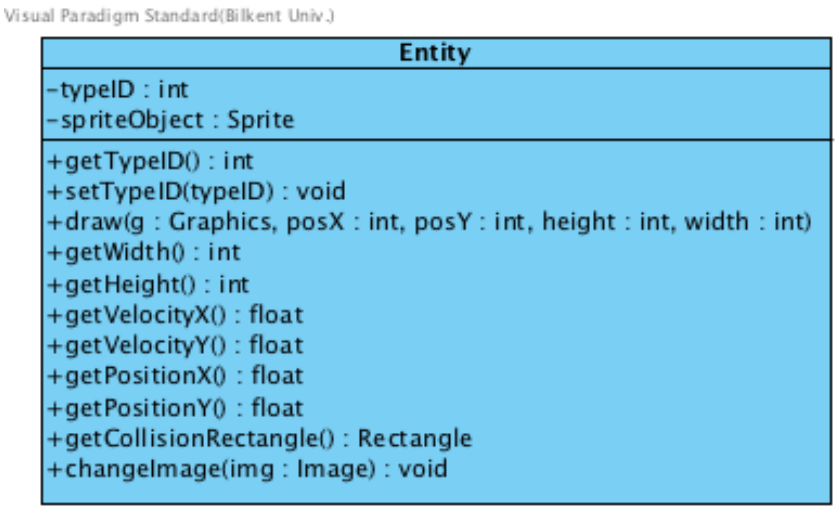
Constructors:

public Sprite (Image img, float posX, float posY, int height, int width): Constructor for Sprite object with given attributes.

Methods:

public void update (long time): This method takes a time parameter and updates Sprite object's position with respect to time parameter. Sprite object's velocity attributes are multiplied with time parameter and added to position parameters to determine new location.

3.4.3 Entity Class



→ Entity is the abstract class and key class for all game objects. Every class from this point on essentially extends this abstract class, i.e. contain this class' attributes and can use this it's methods for core transitions in game. Entity also contains a Sprite object to manipulate in game appereance.

Attributes:

private int typeID: This attribute is for child classes to determine which type of Entity the object is.

private Sprite spriteObject: Sprite instance for enabling movement.

Methods:

public int getTypeID(): Accessor for the typeID attribute.

public int setTypeID (int typeID): Mutator for the typeID attribute.

public void draw (Graphics g, int posX, int posY, int height, int width): This method is to draw Entity itself with given positions for upper left corner and height, width. To enable drawing method also has a Graphics object to enable drawing.

public int getWidth(): Method for returning the width of the image from the game object, which is inside spriteObject attribute.

public int getHeight(): Method for returning the height of the image from the game object, which is inside spriteObject attribute.

public float getVelocityX(): Method for returning the x-component of velocity of the image from the game object, which is inside spriteObject attribute.

public float getVelocityY(): Method for returning the y-component of velocity of the image from the game object, which is inside spriteObject attribute.

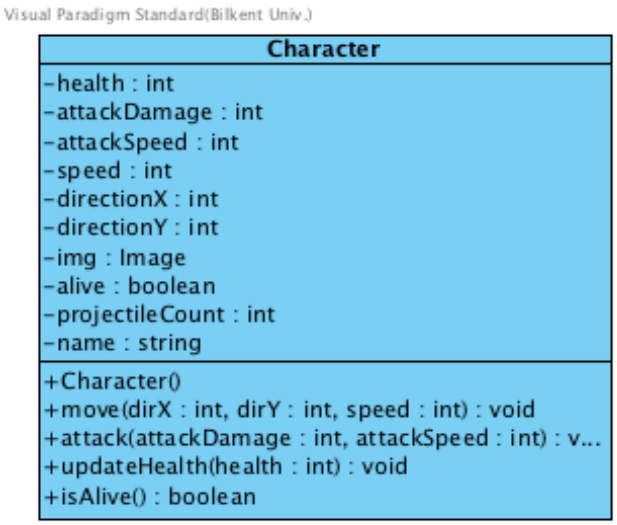
public float getPositionX(): Method for returning the x-component of upper left corner's position of the image from the game object, which is inside spriteObject attribute.

public float getPositionX (): Method for returning the y-component of upper left corner's position of the image from the game object, which is inside spriteObject attribute.

public Rectangle getCollisionRectangle(): Method for returning the boundarie of spriteObject's image attribute.

public void changeImage(Image img): Method for replacing the Entity's image.

3.4.4 Character Class



Attributes:

private int health: Attribute to indicate a character's health.

private int attackDamage: Attribute to indicate a character's attack damage.

private int attackSpeed: Attribute to indicate a character's attack speed.

private int speed: Attribute to indicate a character's move speed.

private int directionX: Attribute to indicate a character's movement direction's x-component.

private int directionY: Attribute to indicate a character's movement direction's y-component.

private Image img: Attribute to indicate a character's health.

private boolean alive: Attribute to indicate if a character is alive or not.

private int projectileCount: Attribute to indicate how many projectiles does a character fire when attack() function is called.

private String name: Attribute to indicate character's name.

Constructor:

public Character (): Default constructors with default attributes.

Methods:

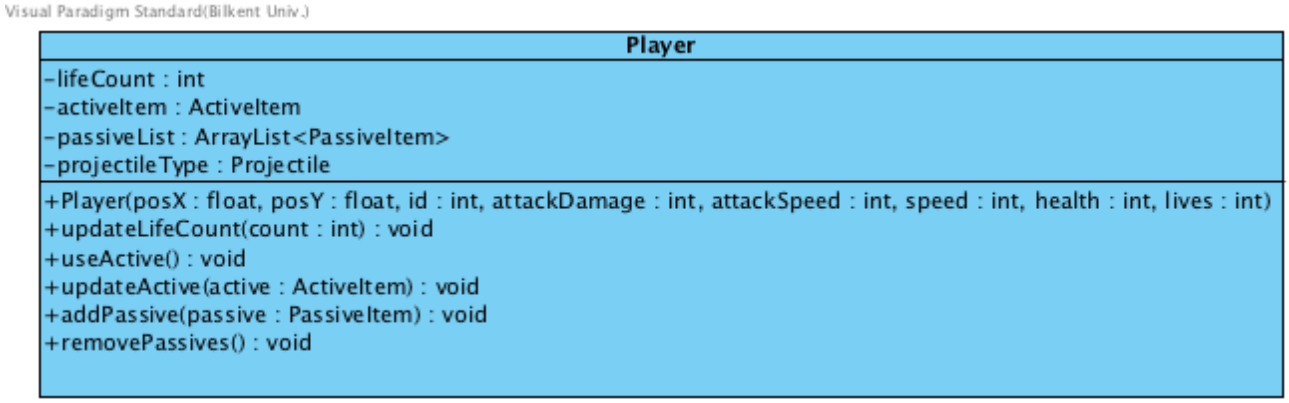
public void move (int dirX, int dirY, int speed): This method takes 3 parameters; 2 int parameters which are x and y components to indicate directions to move to and an int parameter which is the speed attribute of Character object to determine how fast object going to move.

public void attack (int attackDamage, int attackSpeed): This method takes 2 int parameters; attackDamage attribute of Character object and attackSpeed attribute of Character object. When method called character while fire Prjoctile objects which will have the same parameters.

public void updateHealth (int health): This method is to update Character object's health attribute in case of a decrease or increase in health while playing the game.

public boolean isAlive (): This method checks the alive attribute of Character object to determine if character is alive or not.

3.4.5 Player Class



Attributes:

private int lifeCount: This attribute will hold how many lives does Player object has.

private ActiveItem activeItem: Attribute to hold an ActiveItem object for Player object which it can use throughout the game when it posseses one. Initialized to null at the start.

private ArrayList<PassiveItem> passiveList: ArrayList attribute hold the PassiveItem objects of Player Object. Since passive items are permanent when Player gets it, there will be more than one PassiveItem objects of a Player object throughout the game.

private Projectile projectileType: Attribute for attack() method. This attribute will shape the projectile's behavior.

Constructor:

public Player (float posX, float posY, int id, int attackDamage, int attackSpeed, int speed, int health, int lives): Constructor with parameters that come from Entity or Character class except lifeCount attribute.

Methods:

public void updateLifeCount(int count): Method for increasing or decreasing lifeCount attribute.

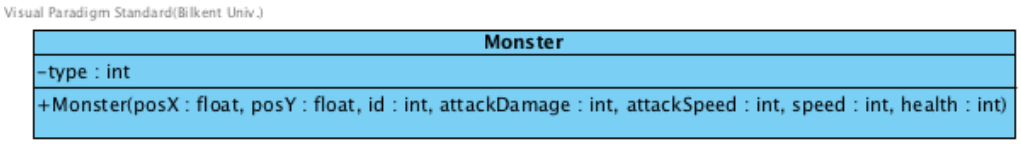
public void useActive(): Method for using ActiveItem object that is inside Player object

public void updateActive(ActiveItem active): Method for either adding or deleting ActiveItem from Player object

public void addPassive(PassiveItem passive): Method for adding a PassiveItem object to PassiveItem ArrayList that is inside Player object.

public void removePassive(): Method for removing a PassiveItem object from PassiveItem ArrayList that is inside Player object.

3.4.6 Monster Class



Attributes:

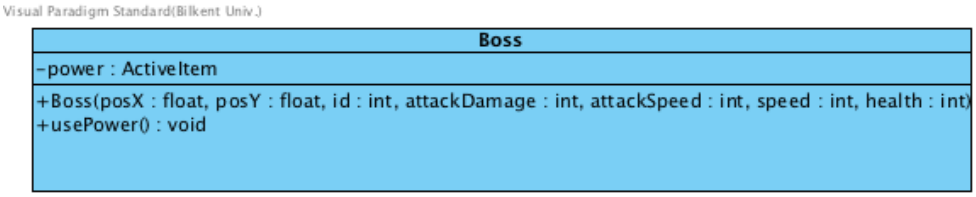
private int type: attribute for determining the type of monster.

Constructor:

Monster (float posX, float posY, int id, int attackDamage, int attackSpeed, int speed, int health): Constructor with position, attack damage, attack speed, speed and health

parameters that come from Character and Entity classes. Additionally, id parameter establishes type of monster.

3.4.7 Boss Class



Attributes:

private ActiveItem power: This attribute holds the power of boss which is also an ActiveItem.

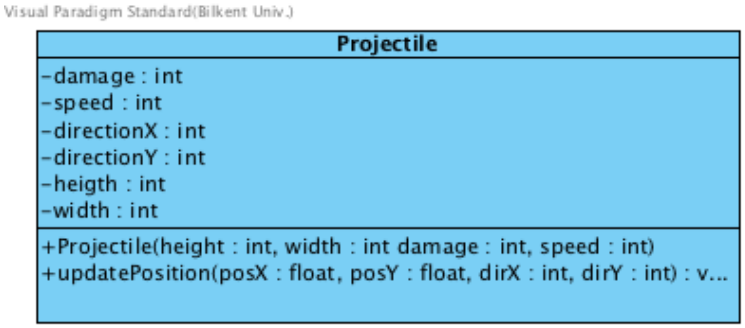
Constructor:

Boss (float posX, float posY, int id, int attackDamage, int attackSpeed, int speed, int health): Constructor with position, attack damage, attack speed, speed, id, health parameters that come from Monster, Character and Entity classes.

Methods:

public void usePower(): This method enables Boss to use its power.

3.4.8 Projectile Class



Attributes:

private int damage: Attribute for holding the value of projectile damage.

private int speed: Attribute for holding the value of projectile's movement speed.

private int directionX: Attribute to indicate projectile's movement direction's x-component.

private int directionY: Attribute to indicate projectile's movement direction's y-component.

private int height: Attribute for holding projectile's image height.

private int width: Attribute for holding projectile's image width.

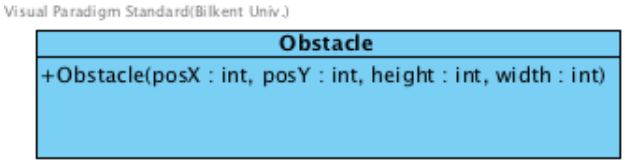
Constructor:

public Projectile (int heigth, int width, int damage, int speed): Constructor with size, damage and speed.

Methods:

public void updatePosition(float posX, float posY, int dirX, int dirY, int speed): This method takes position of left upper corner as two parameters and updates Sprite object's position with respect to direction parameters.

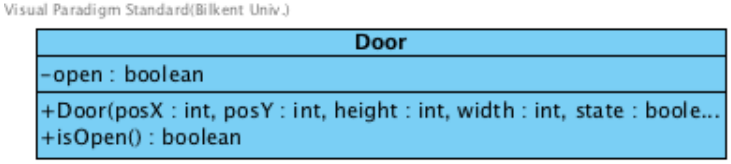
3.4.9 Obstacle Class



Constructor:

public Obstacle (int posX, int posY, int height, int width): Constructor with position and height, width attributes. Position and height - width attributes come from spriteObject which is inside Entity class.

Door Class



Attributes:

private boolean open: Boolean attribute to indicate the state of the door, initialized false at first.

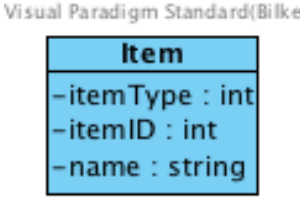
Constructor:

Door(int posX, int posY, int height, int width, boolean state): Constructor with position and height, width attributes and the boolean state. Position and height, width attributes comes from spriteObject which is inside Entity class.

Methods:

public boolean isOpen(): Method for checking the open attribute of Door object to determine if a room is completed and Player has been granted to pass the room.

3.4.10 Item Class



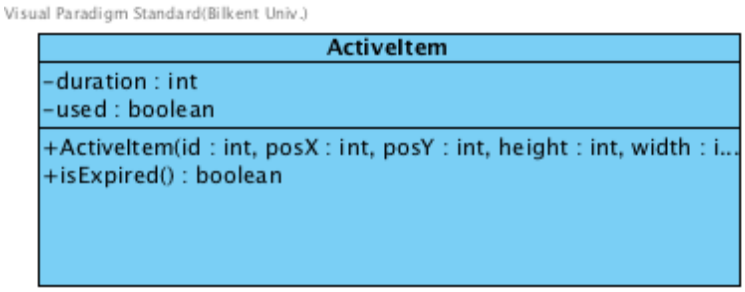
Attributes:

private int itemType: Attribute to indicate itemType whether it is a PassiveItem or ActiveItem.

private int itemID: Attribute to indicate which item it is inside the PassiveItem list or ActiveItem list.

private String name: Attribute for the name of the Item object.

3.4.11 ActiveItem Class



Attributes:

private int duration: Attribute for indicating the expiration duration of ActiveItem

private boolean used: Boolean attribute to enable active item, initialized false when item acquired.

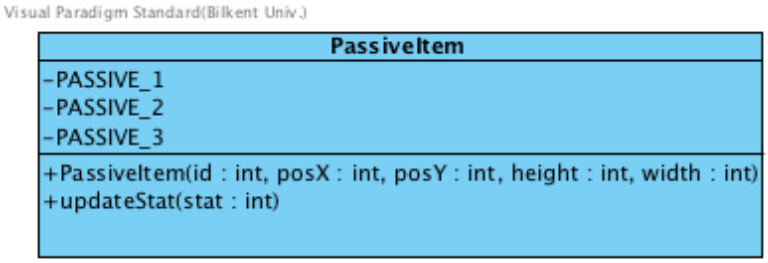
Constructor:

ActiveItem (int id, int posX, int posY, int height, int width): Constructor with an int parameter id to determine type. Position and height, width attributes come from spriteObject which is inside Entity class.

Methods:

public boolean isExpired(): Method for checking the duration attribute of ActiveItem object once it came into use.

3.4.12 PassiveItem Class



Attributes:

private constant int PASSIVE_1:

private constant int PASSIVE_2:

private constant int PASSIVE_3:

→ These attributes are to determine types of passive items

Constructor:

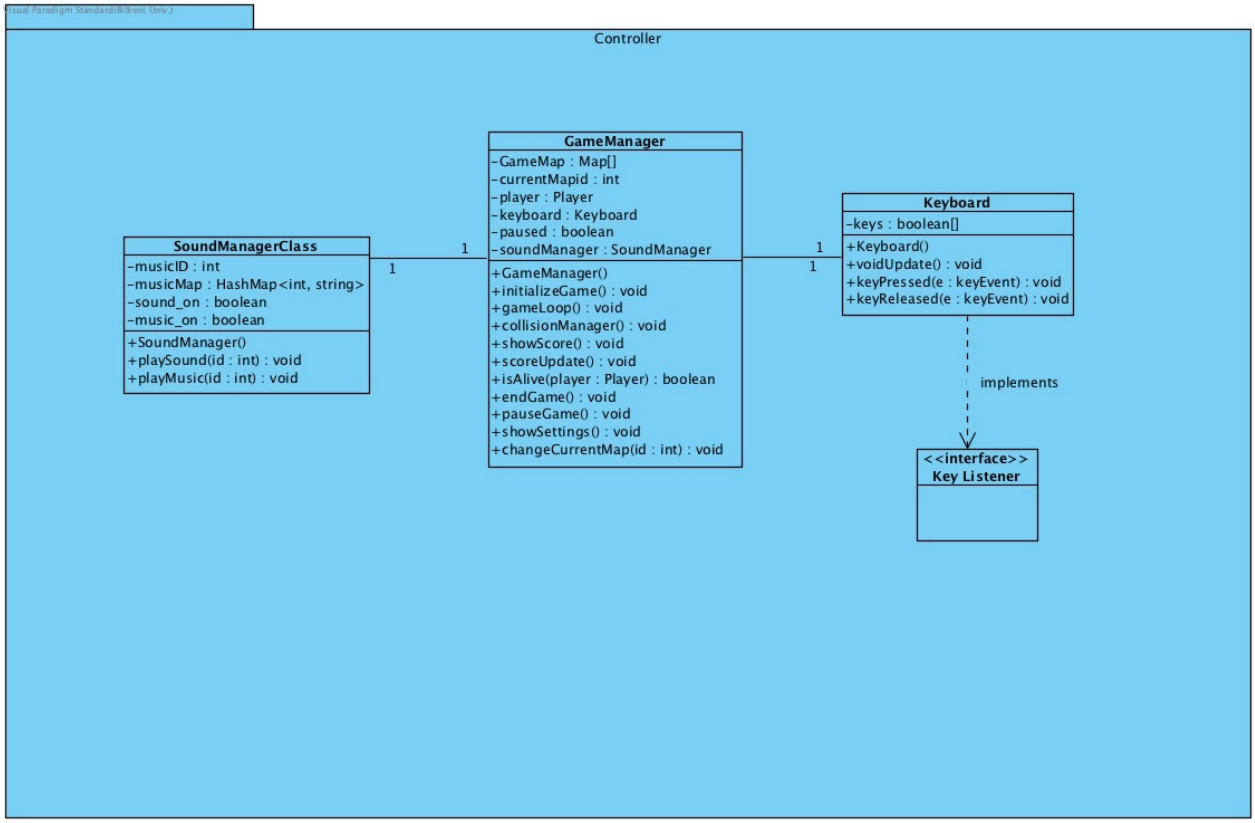
public PassiveItem (int id, int posX, int posY, int height, int width); Constructor with id parameter to determine type. Position and height, width attributes come from spriteObject which is inside Entity class.

Methods:

public void updateStat(int stat): Method for adding the stat upgrade to player taking an int parameter to determine amount of upgrade.

3.5 Game Controller Subsystem

Game Controller subsystem will be the inter-layer in three-tier system which will contain necessary classes to manage user input, game sounds and detect any change in game state. The most important class in this subsystem is Game Manager class has access to all the classes in the subsystem. Game Manager class has the gameLoop() method which runs the game until there is an interruption or the player dies. Keyboard class uses KeyEvent interface and detects any user input and GameManager uses this data from the Keyboard class. SoundManager class plays or disables any sound and music in the game.



It is also controlled by GameManager class which manipulates it with respect to the user input. GameManger has access to the model and view subsystems of the game as well. Basically, it is the heart of our game.

3.5.1 Game Manager Class

Game Manager class is the main class which runs the game. Game loop runs in this class. It performs the actions requested from users. The state of game is determined by this class, for example it determines the game if it is in pause state or run state.

Atttibutes:

private GameMenu gameMenu: This is initialized when game runs. It offers players the menu view of the game which they can start the game or change the settings, or look at high scores, help, credits etc.

private boolean paused: This determines the state of the game. It can be paused or running.

private GameMap gameMap[ ]: This is an array of Map objects. It contains the maps which are the layers in the game. Every map contains the rooms where player walks. Shortly it is the game plan which enables the game manager to manage game mechanics in the model subsystem.

private Player player: This is player object. It is initialized when game initializes. Game Manager class uses player object to manage its actions. Game manager is like a bridge between the room and the player. Everytime player changes a state game manager updates the rooms conditions and etc.

private int currentMapID: Game manager uses this integer value to determine the current map that the player is inside of.

private SoundManager soundManager: Sound Manager object enables Game Manager to control sounds and music in game. Everytime a settings is selected Game Manager passes this information to Sound Manager class to change the sound.

private Keyboard key; Keyboard object enables Game Manager to control key actions and game mechanics in game.

Constructor:

public GameManager(): It initializes the attributes and some other necessary functions. It initializes Game Manager class.

Methods:

public void gameLoop(): This method runs as long as player plays the game and updates continuously. It checks all the changes of the game.

public void initilizeGame(): If user presses the play button, this method initializes the game sets de.

public void manageCollision(): This method invoked by the keyboard press of the player. It calls the getCurrentRoom() method of Map class. Then, it uses the checkCollision method of the Room. Room returns an object to Game manager class. Game Manager checks if this object is null or not. If the object is null, player can move and update its position. Otherwise, if the object is an Item, GameManager class calls addItem() method of the Player and player gets upgrade. If the object is an Obstacle, then player can not move. Basically, this method manages the movement of the player.

public void isAlive(): Checks if player has health.

public void endGame():This method uses isAlive() method to check if the player can still play the game. If the player has no health left, then the Game Manager finishes the game.

public void pauseGame(): This method is invoked by the user keyboard input if the pause key is pressed. Game Manager uses this method to stop the game and using the Game Menu object it makes the menu appear on the screen.

public void showSettings(): This method is invoked by the player input and it displays the sound settings on the screen by using Game Menu object.

public void changeCurrentMap(): This method is used when GameManager detects wether the player passed to another map(layer) and updates the view and the generates the new map.

3.5.2 Sound Manager Class

This class is for controlling sound effects and music in game. Game Manager class uses this class as an attribute and together with keyboard class, Game Manager changes the sound states according to user input.

Atttibutes:

private boolean sound_on: This attribute is for enabling sound effects in game or disabling it.

private boolean music_on: This attribute is for enabling music in game or disabling it.

private int soundMap<int,string>: This map hashes id's of sounds with respect to their name.

private int musicMap<int,string>: This map hashes id's of map's current id with respect to their names.

Constructor:

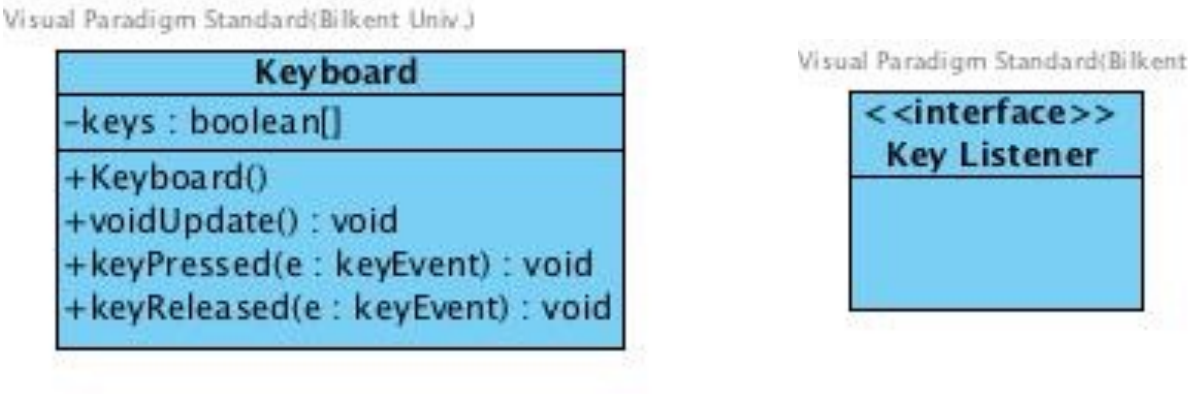
public SoundManager(); initiliazes object, sound_on and music_on boolean is true by default.

Methods:

public void playSound(int soundID); Takes sound id for example collision sound, projectile sound etc and plays it by using the musicMap which holds names of sound.

public void playMusic(int musicID); Takes current map's musicID and plays it.

3.5.3. Keyboard Class



This class enables Game Manager to get keyboard inputs from player. This class implements KeyListener interface. Moreover, Game Manager uses this class to manage view and model subsystems in order to change sound settings, show menu, move player, attack and etc.

Attributes:

Private boolean keys;

Constructor:

Public Keyboard(); Initializes the keyboard object for game. The keys are false by default.

Methods:

Public void keyPressed(KeyEvent e); Determines pressed key.

Public void keyReleased(KeyEvent e); Determines released key.

4. Low-level Design

4.1 Object Design Trade-Offs

- **Efficiency vs. Portability:** Our game needs to be implemented efficiently in order to give the enjoyable experience we intend to give to the user while playing the game since the game is fast paced. We can afford to sacrifice portability since the game is intended to be a desktop/ windows game, so we don't need to think about portability and how to port the game to other systems such as game consoles or other operating systems.

- **Development Time vs. User Experience:** Since we have to keep up with the deadlines, we are sacrificing possible functionalities for rapid development. Our game could have been more complex in the sense of functionality such as different mechanisms like jumping over obstacles or shopkeepers where the user could buy upgrades or secret rooms that could be reached by completing different tasks.

- **Cost vs. Robustness:** Our team is consisted of 4 people so the cost is very low. By having few people working on the project, robustness of it is sacrificed so even if we try hard, there will most probably be some bugs and unintended features in the game.

5. Glossary & References

[1] https://en.wikipedia.org/wiki/The_Binding_of_Isaac_(video_game)

[2] https://en.wikipedia.org/wiki/FPS

[3] https://en.wikipedia.org/wiki/User_interface

[4] https://en.wikipedia.org/wiki/Roguelike