CS 319 - Object-Oriented Software Engineering

Design Report Iteration 2

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. Moreover, in terms of boundary conditions such as power outages, the highscores will not be lost to keep persistent data.

- **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

2.1.1 Subsystem Decomposition

We have decomposed our system using the MVP software architecture style which is an improved version of MVC(Model/View/Controller) that subsystems consists of three types: Model, View and Presenter. Our game is heavily dependent on game logic, user interface and the interaction between them so we thought that MVP architecture style fits our game better. View is the top layer and it will act as an interaction between the user and the system and it will depend on presenter. Presenter is the middleware which connects Model and View, updates Model when

an event fired and renders new model into the View. Model contains the entity objects which are mostly application domain objects. Model consists of the data structure of our entire game. Model is the bottom layer and it will consist of handling the data flow of the game. By following MVP we are aiming to create a system with high coherence and low coupling. We used opaque layering as a layering method.

2.1.2 Design Patterns

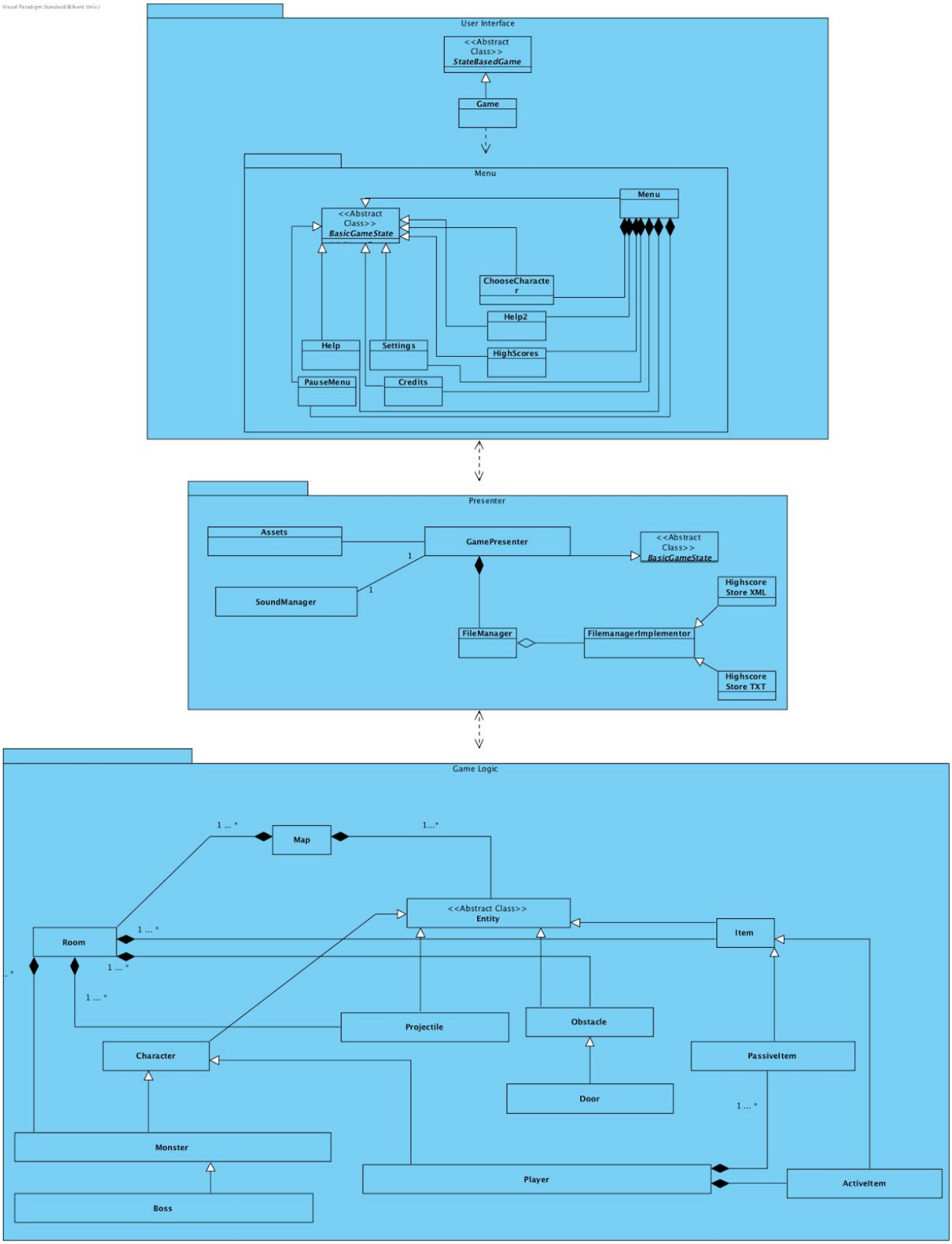
We also made use of several design patterns including State Based design, Facade Design, Bridge Design and Composite Design patterns. Design patterns allowed us to solve specific solution domain related problems.

State Based design was used in the View layer of the game. It allows easy transaction and separation between different states of the game. Moreover, it enables us to separate the actual game content and menu related content.

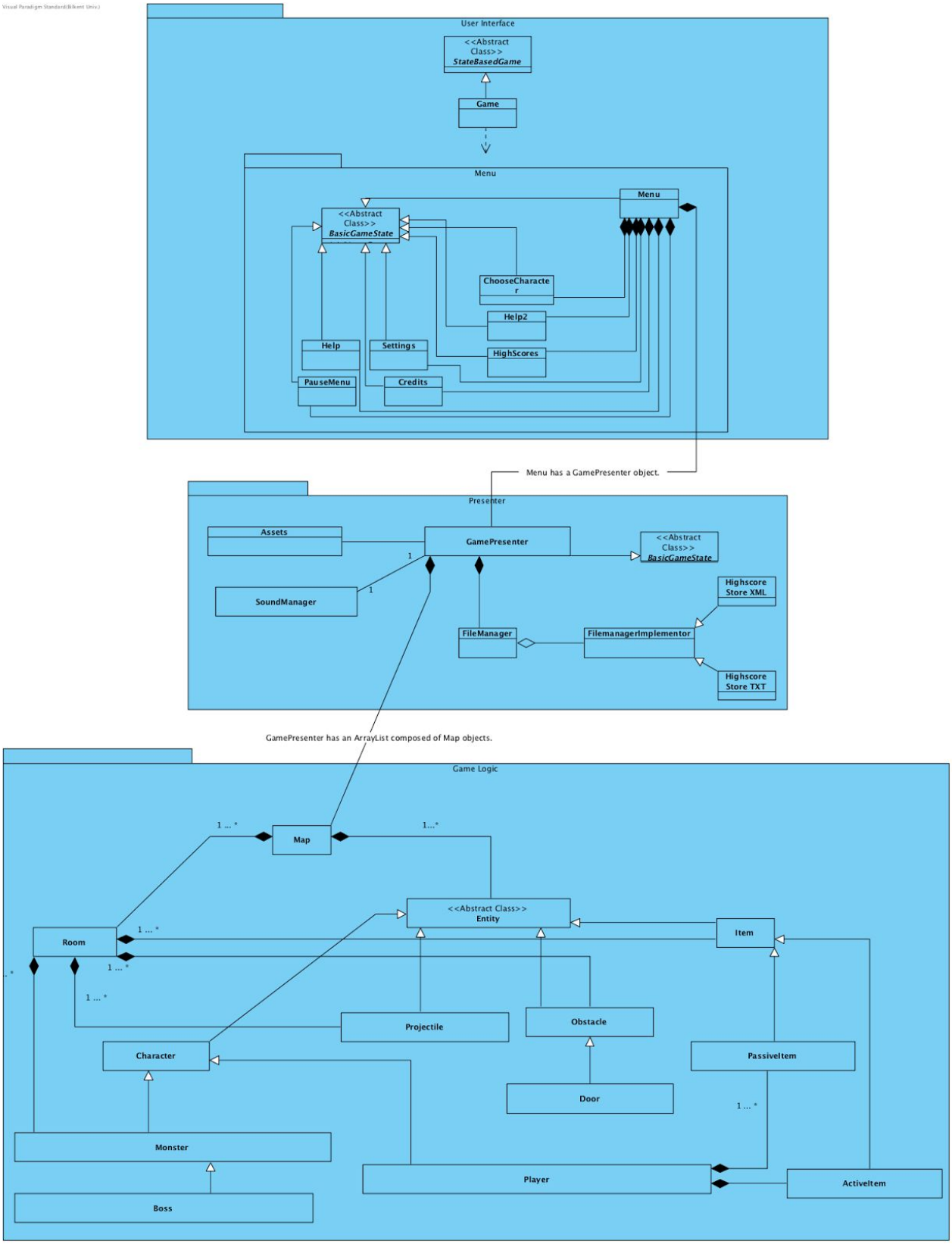
Facade design was used in Model layer. Map Class is used as the Facade class which serves as the interface of the Model layer which is used by Presenter in the architectural style. Facade design decreases the coupling between the Presenter layer and the Model layer.

Bridge design was used in Presenter layer. This design allowed us to abstract model from the real implementation of file manager that we can test and try different implementations for storing highscore data. This pattern allowed us to decouple *FileManager* class and *FileManagerImplementor* classes so that former they provide different levels of functionality. *FileManagerImplementer* is the parent class of Highscore XML Store and Highscore TXT Store which provide different low-level implementations for storing highscore datas.

Basic Subsystem Decomposition

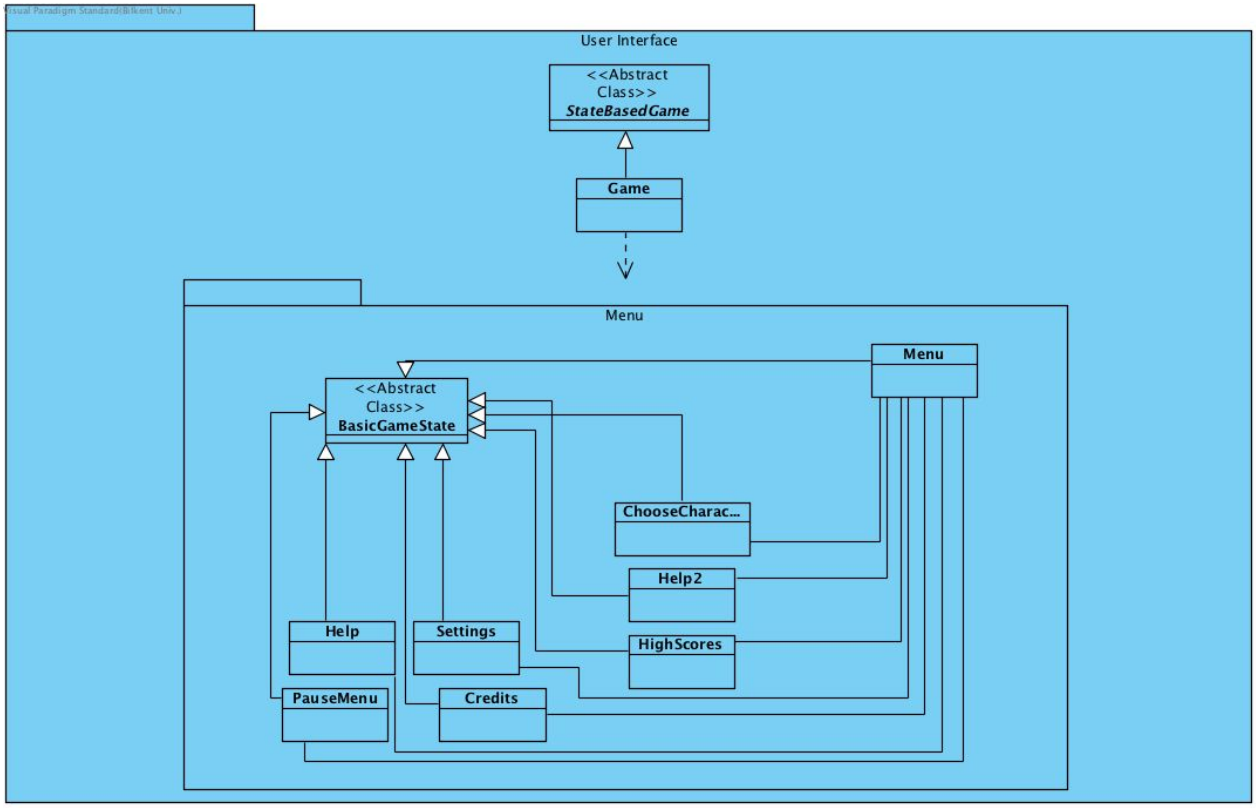


Detailed Subsystem Design

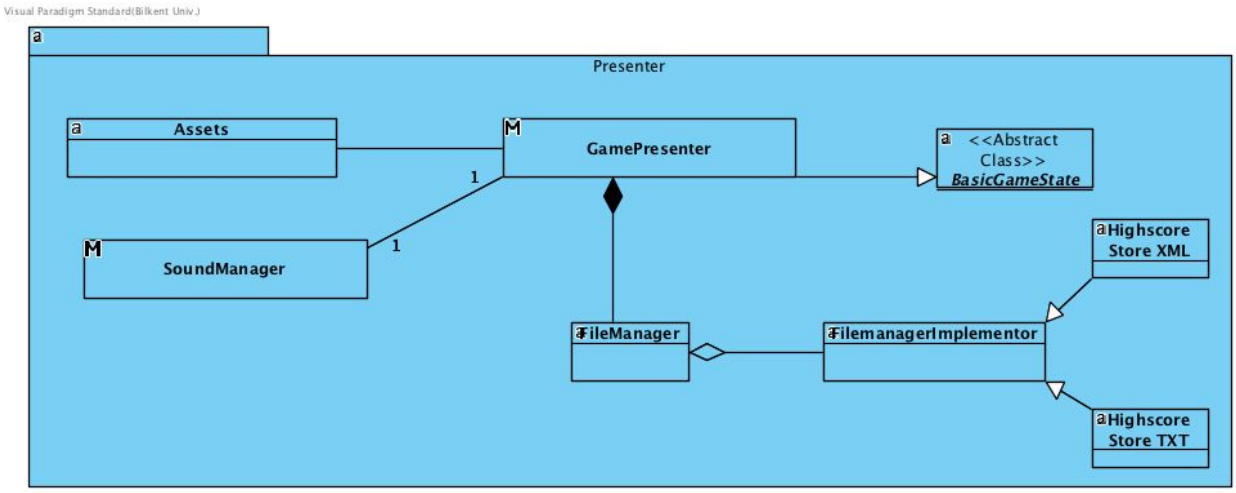


High Level View of Subsystem Decomposition

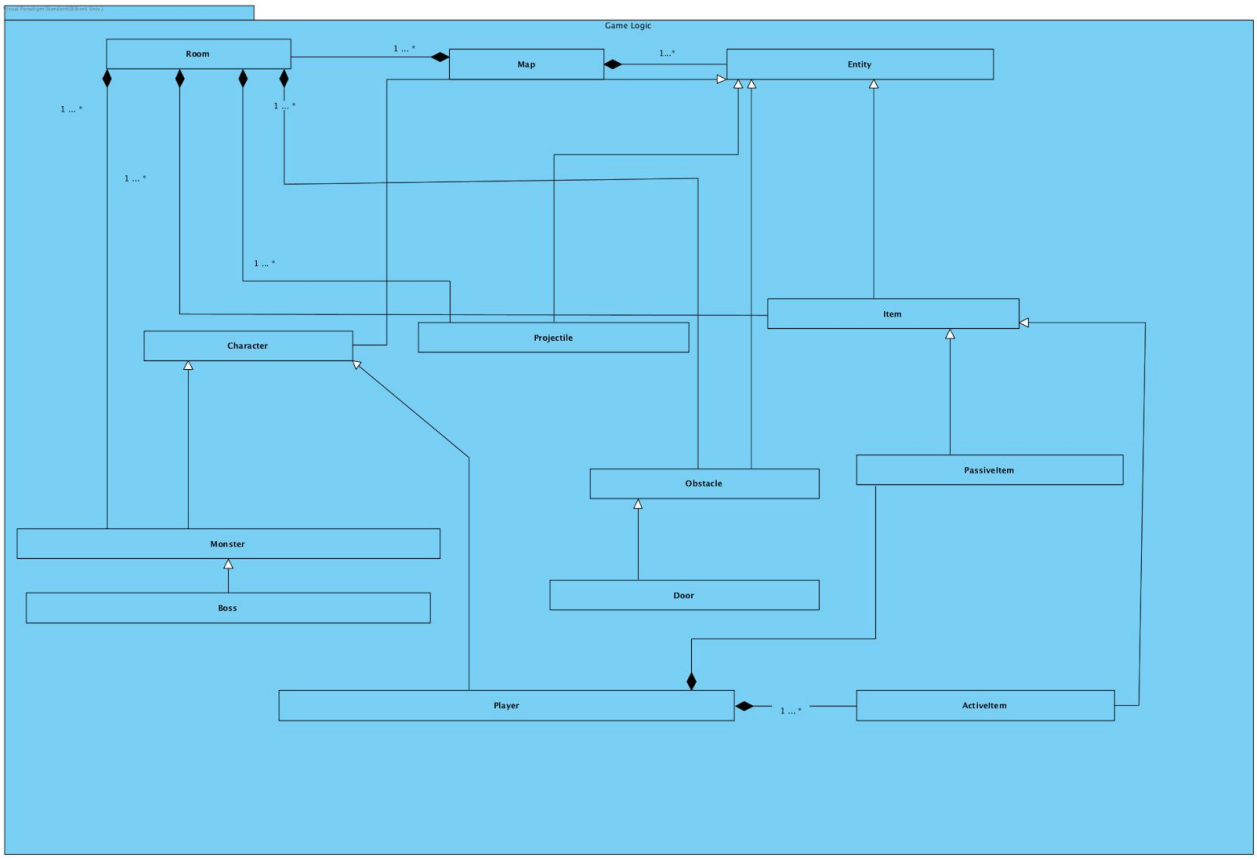
User Interface Subsystem



Presenter 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

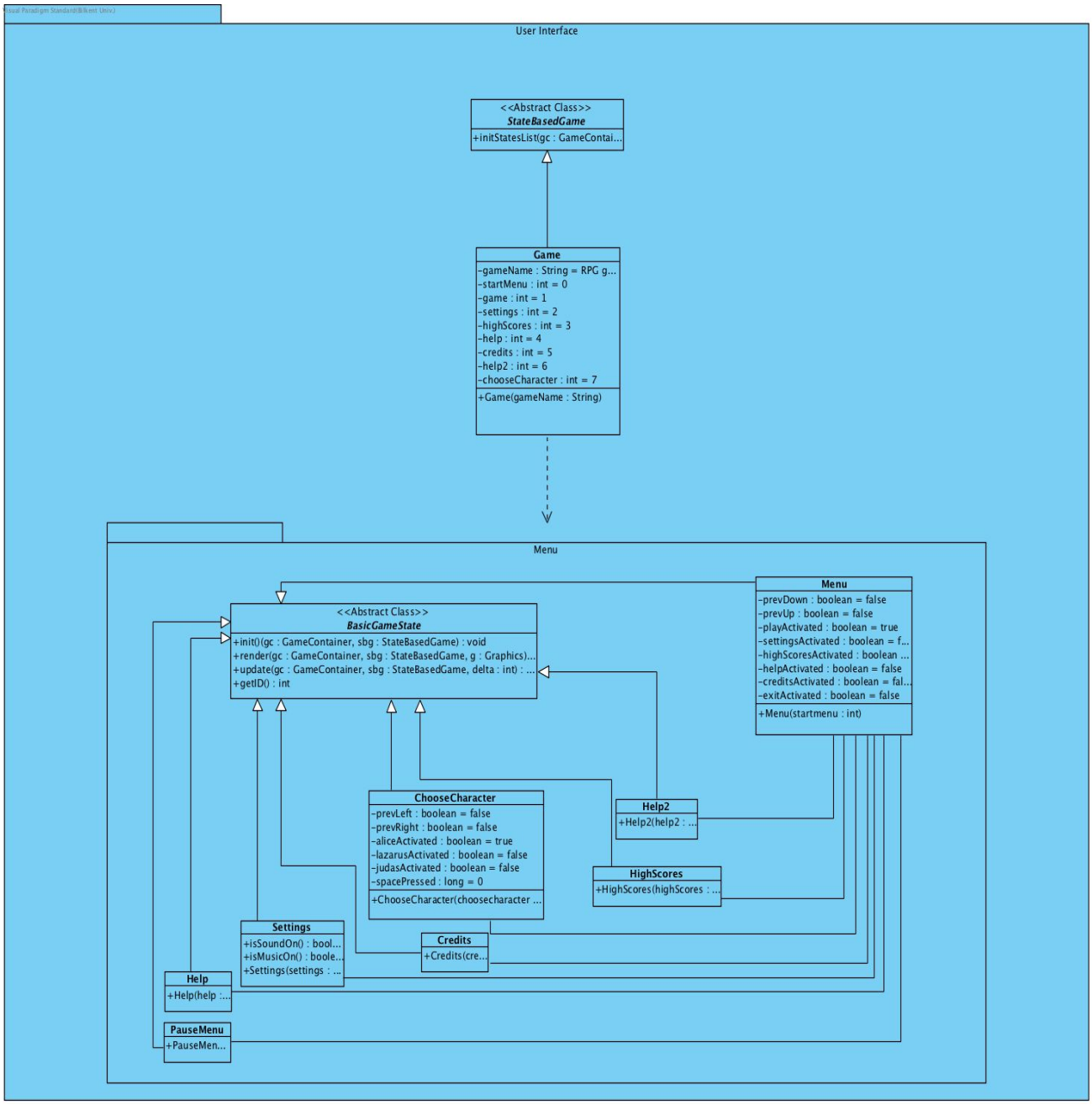
The maps will have boundaries on the edges, the implementation will include cases where the player tries to go beyond these boundaries and the game will not allow going beyond the boundaries. We will still catch exceptions about boundaries even if we have checks controlling the boundaries of the map 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 player's 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. Moreover, game starts to save high scores every 10 seconds in order to avoid data loss during power outages. Persistent storage of high scores are important since player competes with their previous scores every time they start the game.

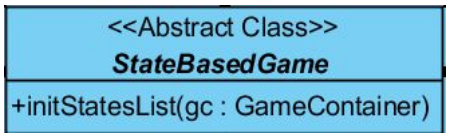
3. Subsystem Services

3.2. User Interface Subsystem

The interaction between the user and the system is provided by the User Interface subsystem. This subsystem makes use of the State Based Design Pattern. We have used game states to provide efficient way of maintaining and testing different components of our game. We believe that this design pattern have become helpful for lessening the coupling between the View and Presenter. This interaction is done with the use of graphical off the shelf components. The User Interface subsystem consists of 7 classes which are Menu which contains PauseMenu, Settings, Help, Help2, Credits, ChooseCharacter. Game class extends StateBasedGame and contains the Menu class, handles transitions over frames.


User Interface
<<Abstract Class>>
StateBasedGame
+initStatesList(gc : GameContai...
Game
-gameName : String = RPG g...
-startMenu : int = 0
-game : int = 1
-settings : int = 2
-highScores : int = 3
-help : int = 4
-credits : int = 5
-help2 : int = 6
-chooseCharacter : int = 7
+Game(gameName : String)
Menu
<<Abstract Class>>
BasicGameState
+init()(gc : GameContainer, sbg : StateBasedGame) : void
+render(gc : GameContainer, sbg : StateBasedGame, g : Graphics)...
+update(gc : GameContainer, sbg : StateBasedGame, delta : int) : ...
+getID() : int
Menu
-prevDown : boolean = false
-prevUp : boolean = false
-playActivated : boolean = true
-settingsActivated : boolean = f...
-highScoresActivated : boolean ...
-helpActivated : boolean = false
-creditsActivated : boolean = fal...
-exitActivated : boolean = false
+Menu(startmenu : int)
ChooseCharacter
-prevLeft : boolean = false
-prevRight : boolean = false
-aliceActivated : boolean = true
-lazarusActivated : boolean = false
-judasActivated : boolean = false
-spacePressed : long = 0
+ChooseCharacter(choosecharacter ...
Help2
+Help2(help2 : ...
HighScores
+HighScores(highScores : ...
Settings
+isSoundOn() : bool...
+isMusicOn() : boole...
+Settings(settings : ...
Credits
+Credits(cre...
Help
+Help(help :...
PauseMenu
+PauseMen...

3.2.1. StateBasedGame

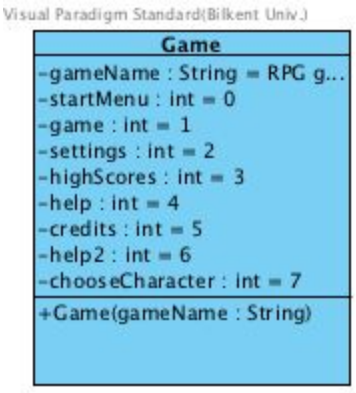


StateBasedGame class is the parent class of Game class. It is an abstract class so there are no StateBasedGame objects in the system, it is only used for its methods and properties.

Methods

public void initStatesList(GameContainer gc): Takes a GameContainer object named gc to initialize other states that will be used in the system.

3.2.2. Game



Attributes:

private String gameName: This attribute holds for game Name

private int startMenu: This attribute holds for start menu. 0 in our code

private int game: This attribute holds for game. 1 in our code

private int settings: This attribute holds for settings option. 2 in our code

private int highScores: This attribute holds for seeing high scores. 3 in our code

private int help: This attribute holds for help section. 4 in our code.

private int credits: This attribute holds for credits section in menu. 5 in our code

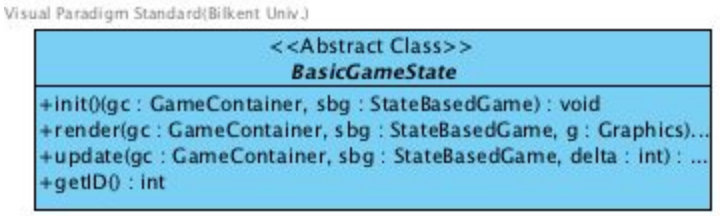
private int help2: This attribute holds for another help section in our code. 6 in our code.

private int chooseCharacter: This attribute holds for choosing character section in menu. 7 in our code

Methods:

private void Game(string gameName): Takes the game name and creates a game. Adds states to the game. For example, each attribute and its number represents a state. Again for example it adds setting class as a state to this game etc.

3.2.3. Basic Game State



This class is for displaying menu into the screen. It has no attributes.

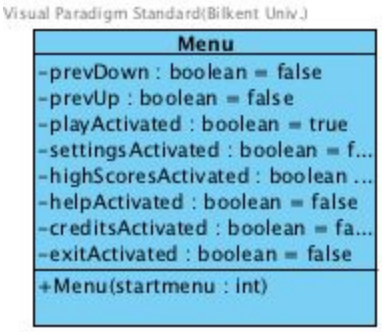
Methods:

public void init(GameContainer gc, StatebasedGame sbg): Draws initial state of menu into the screen. Only initializes when menu is initialized.

public void render(GameContainer gc, StatebasedGame sbg, Graphics g): This method draws menu constantly on the screen.

public void update(GameContainer gc, StateBasedGame sbg, int delta): This method updates the logic behind it. In other words for example when user presses down, it goes to the option behind current option and this updates model according to it and then render function re-draws onto screen.

3.2.4. Menu



The Menu class presents 6 choices unique to the MainMenu class when the user is in the main menu. These choices are: viewHelp(), viewHighScore(), viewCredits() and startGame() viewHelp2(), viewSettings(). startGame() method initiates the game. This class extends BasicGameState class.

Attributes:

private boolean prevDown: Boolean attribute for previous down.

private boolean prevUp: Boolean attribute for previous up.

private boolean playActivated: This attribute holds for if play option is activated or not.

private boolean settingsActivated: This attribute holds for if settings option is activated or not.

private boolean highScoresActivated: This attribute holds for if highscores option is activated or not.

private boolean helpActivated: This attribute holds for if help option is activated or not.

private boolean creditsActivated: This attribute holds for if credits option is activated or not.

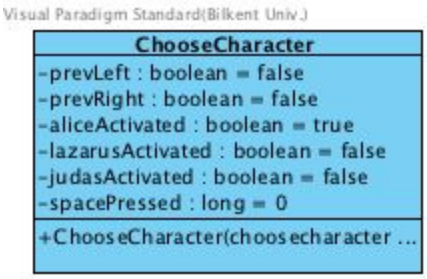
private boolean exitActivated: This attribute holds for if exit option is activated or not.

private boolean chooseCharacterActivated: This attribute holds for if choose character option is activated or not.

Constructor

Menu(int startMenu): Constructor for initalizing start menu.

3.2.5. Choose Character



This class is for players to select the character which they want to play. This class extends BasicGameState class.

Attributes:

private boolean prevLeft: This attribute is for determining user pressed left.

private boolean prefRight: This attribute is for determining if user pressed right.

private boolean aliceActivated: This attribute is for is character Alice is selected or not

private boolean lazarusActivated: This attribute is for is character Lazarus is selected or not

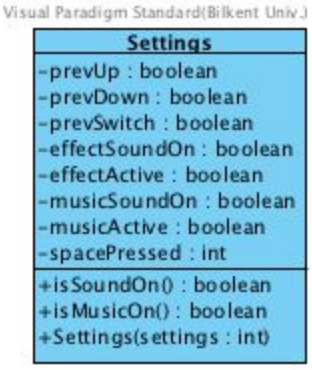
private boolean judasActivated: This attribute is for is character Judas is selected or not

private long spacePressed: This attribute is for is space pressed or not.

Constructor:

ChooseCharacter(): Constructor for choose character class.

3.2.6. Settings Class



This class is for changing settings. It can be accessed from main menu. User's have option to activate sound and music for the game. This class extends BasicGameState class.

Attributes:

private boolean prevUp: Boolean attribute for previous up

private boolean prevDown: Boolean attribute for previous down

private boolean effectSoundOn: For determining if sound on or not.

private boolean musicSoundOn: For determining if music on or not.

Constructor:

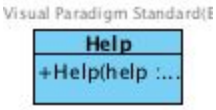
Settings(): Constructor for settings class.

Methods:

public boolean isSoundOn(): Returns the variable isSoundOn.

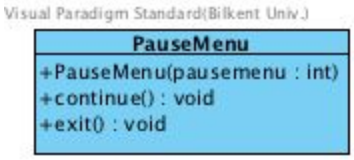
public boolean boolean isMusicOn(): Returns the variable isMusicOn.

3.2.7. Help Class



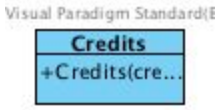
This class is for Help option in main menu. User's can see key bindings of the game and get help about game. This class extends Basic Game State class.

3.2.8. Pause Menu



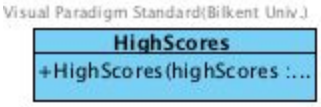
Pause Menu class has the common methods with the menu class but instead of play game it has continue game option for continuing game .

3.2.9. Credits



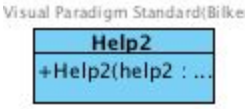
This class is for Credits section in main menu. This class also extends Basic Game State class too.

3.2.10. HighScores



This class is for keeping highscores of users. This class also extends Basic Game State class

3.2.11. Help2

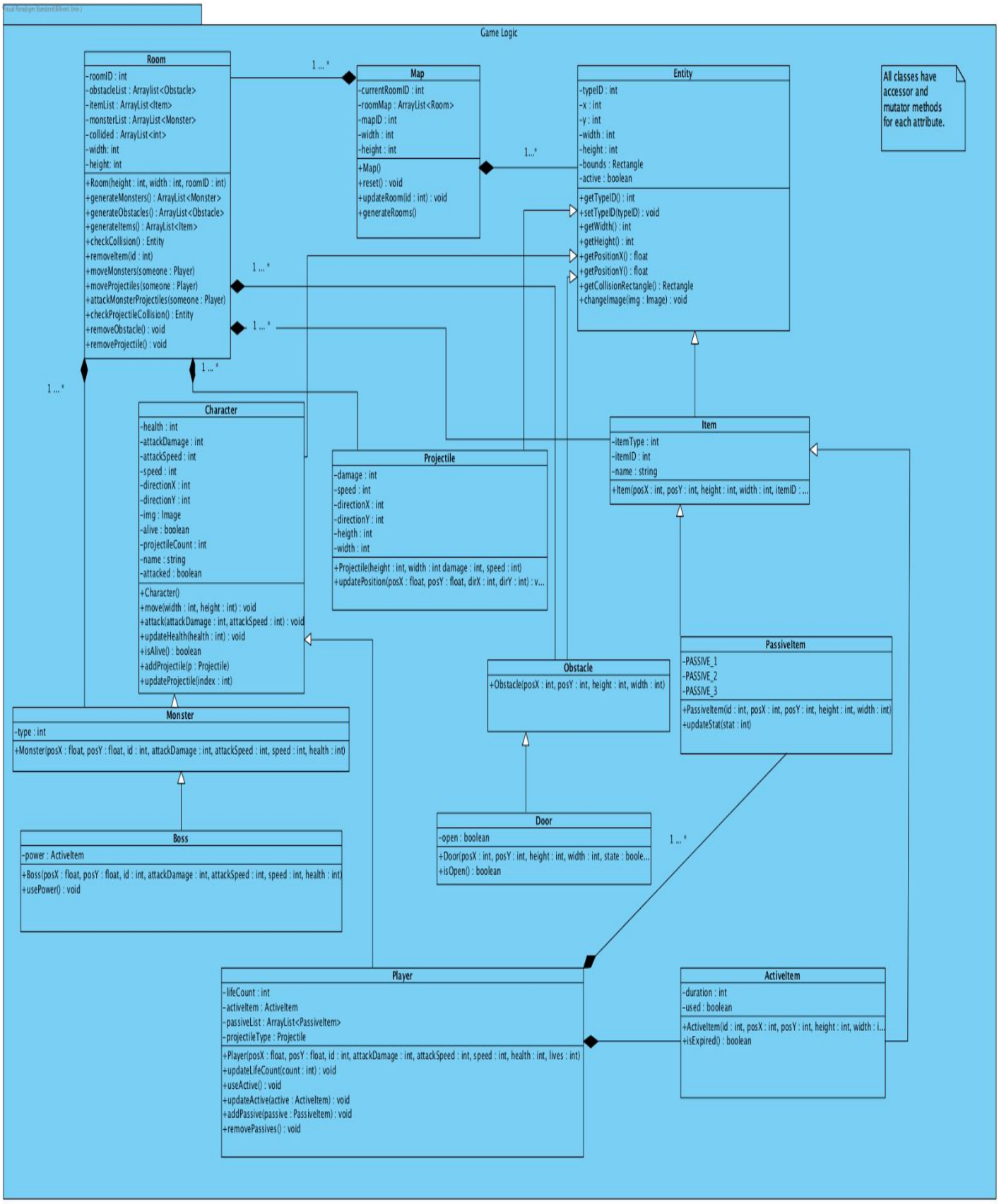


This class is for second Help option in main menu. This class extends Basic Game State class.

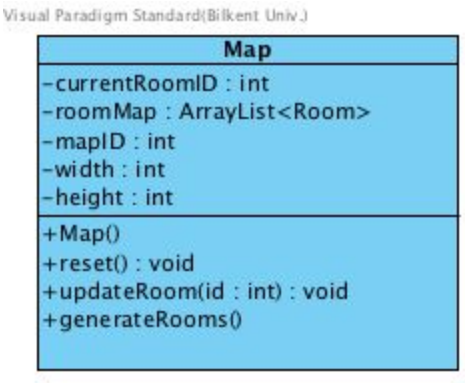
3.3. Game Logic Subsystem Interface

Game Logic subsystem will be the “model” in MVP which will contain the key components of the product. Key class in this subsystem is Entity class which is an abstract class for nearly every class in following subsystem. Entity class is the keystone for 10 classes in the subsystem. Moreover, there is Room class and Map class for arranging each room component inside the game map. There are also compositions between Map – Room, Map – Entity.

We have made use of Facade design pattern and used *Map* class as Facade class which serves as interface of the Model layer of the game. Basically, *Map* encapsulates the Model subsystem and serves as representer class of the subsystem. Facade design enables to reduce coupling between the Model and Presenter layers. Presenter layer does not have information about the inner dynamics of Model layer. If there is a change in Model Presenter class is not affected but Facade class(*Map)* is affected.


Game Logic
Room
-roomID : int
-obstacleList : Arraylist<Obstacle>
-itemList : ArrayList<Item>
-monsterList : ArrayList<Monster>
-collided : ArrayList<int>
-width: int
-height: int
+Room(height : int, width : int, roomID : int)
+generateMonsters() : ArrayList<Monster>
+generateObstacles() : ArrayList<Obstacle>
+generateItems() : ArrayList<Item>
+checkCollision() : Entity
+removeItem(id : int)
+moveMonsters(someone : Player)
+moveProjectiles(someone : Player)
+attackMonsterProjectiles(someone : Player)
+checkProjectileCollision() : Entity
+removeObstacle() : void
+removeProjectile() : void
1 ... *
Map
-currentRoomID : int
-roomMap : ArrayList<Room>
-mapID : int
-width : int
-height : int
+Map()
+reset() : void
+updateRoom(id : int) : void
+generateRooms()
1...*
Entity
-typeID : int
-x : int
-y : int
-width : int
-height : int
-bounds : Rectangle
-active : boolean
+getTypeID() : int
+setTypeID(typeID) : void
+getWidth() : int
+getHeight() : int
+getPositionX() : float
+getPositionY() : float
+getCollisionRectangle() : Rectangle
+changeImage(img : Image) : void
All classes have accessor and mutator methods for each attribute.
Character
-health : int
-attackDamage : int
-attackSpeed : int
-speed : int
-directionX : int
-directionY : int
-img : Image
-alive : boolean
-projectileCount : int
-name : string
-attacked : boolean
+Character()
+move(width : int, height : int) : void
+attack(attackDamage : int, attackSpeed : int) : void
+updateHealth(health : int) : void
+isAlive() : boolean
+addProjectile(p : Projectile)
+updateProjectile(index : int)
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...
Item
-itemType : int
-itemID : int
-name : string
+Item(posX : int, posY : int, height : int, width : int, itemID : ...
Monster
-type : int
+Monster(posX : float, posY : float, id : int, attackDamage : int, attackSpeed : int, speed : int, health : int)
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)
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
1 ... *
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

3.3.1. Map 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 ArrayList<int> roomMap: This is for holding rooms that exist in Map.

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

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

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

Constructor:

Map(): Default constructor with default attributes.

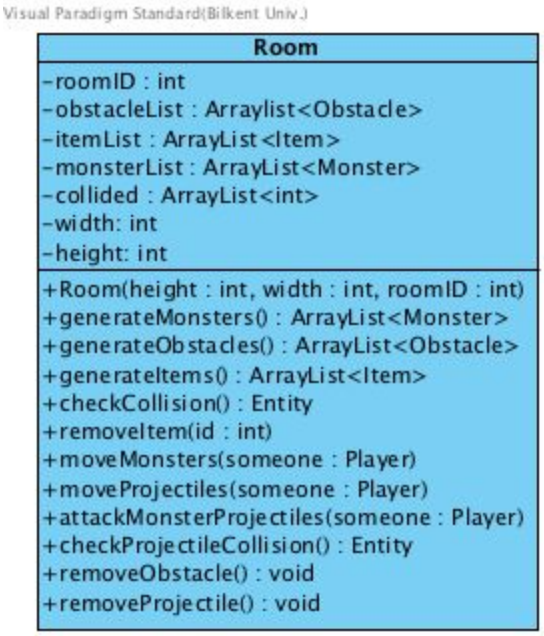
Methods:

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.

public void generateRooms(): This method is for generating a random room in our game since a map is consisted of plenty of rooms.

3.3.2 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<int> collided: This is for every monster. 0 means monster is not collided. If it is bigger than 0, it means it collides to something. It can have value up to 100. This allows monsters to go back when they collide some obstacle or item etc.

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 checkCollision(character Someone): This method checks if there is any collision with player and other entities 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. If there is no collision it returns null.

public Entity checkProjectileCollision(Projectile projectile): Checks if projectiles collides to an entity or not. For example it checks if players attack hit to monster or not. This method returns the collided object. If there is none it returns null.

public void attackMonsterProjectiles(Player someone): This method allows some monsters to attack to our player. Traverses the monster list, if the monster type is an attacking monster, this method makes that monster to attack according to some mathematical calculations using players coordinates.

public void moveProjectiles(Player player): This method updates all projectiles in the current room. If player's attack collides to a creature, this removes the projectile and makes another method call to decrease collided monsters health.

public void moveMonsters(Player someone): This method allows monsters to move through the room. It traverses monsters list and if monster type is a moving monster, it makes it move according to some mathematical calculations according to our players coordinates. It basicly follows our players location.

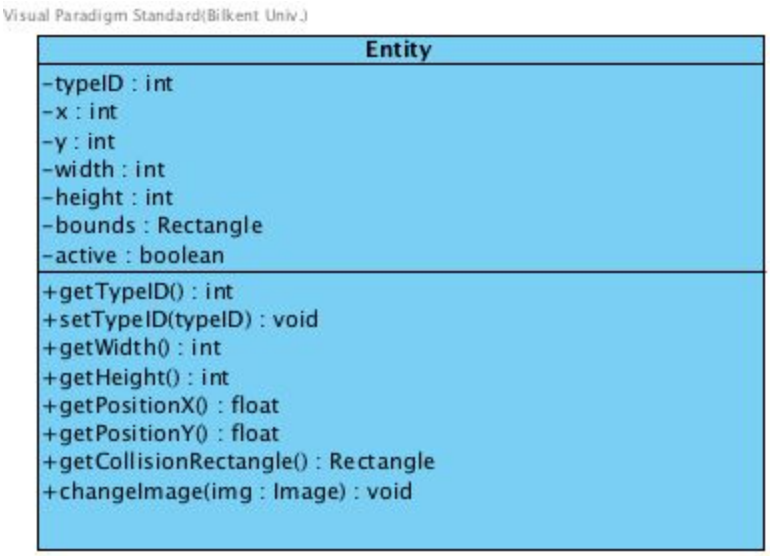
public void removeItem (int id): This method removes the item with the given id from the itemList inside the Room class.

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

3.3.3 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.

Attributes:

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

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

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

private Rectangle bounds: bounds to determine boundaries of object, used in collision detection

private boolean active : This attribue is for visibility of entity.

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 Entity class.

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

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 Entity class

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 Entitiy class attribute.

public Rectangle getCollisionRectangle(): Method for returning the boundary of Entity Object.

3.3.4 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.

protected ArrayList<Projectile> projectile: This holds the projectile list of character.

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.

protected boolean attacked: If player is attacked, this becomes true.

Constructor:

public Character (int x, int y, int typeID,int width, int height, int health , int speed, int attackDamage, int attackSpeed): Constructor with default attributes.

Methods:

public void move (int roomWidth, int roomHeight): This method takes room width and height for keeping characters which can be monsters and players in the room boundary. Basicly this method uses the attributes of characters 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.

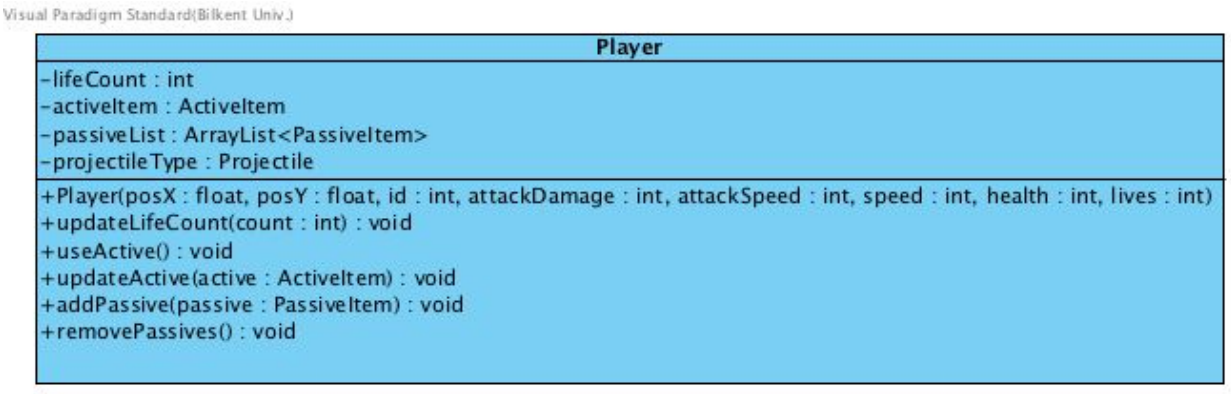
public void updateHealth (int health): This method is to update Character object's health attribute in case of a decrease or increase in health while playing the game.

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

public void addProjectile(): This allows to add a projectile to projectile list.

public void updateProjecile(): This allows to update projectiles location or exsistence etc.

3.3.5 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.

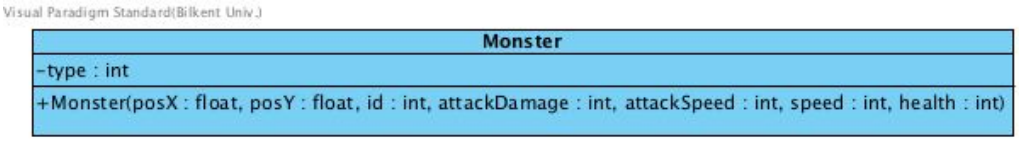
public void useActive(): Method for using ActiveItem object that is inside Player object

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.

3.3.6 Monster Class



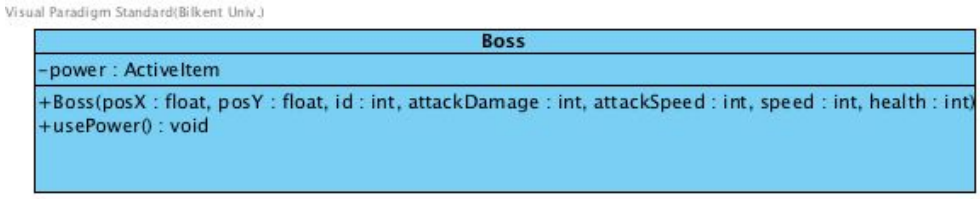
Attributes:

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

Constructor:

Monster (int x, int y, int typeID,int width, int height,int monsterType): 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.

3.3.7 Boss Class



Attributes:

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

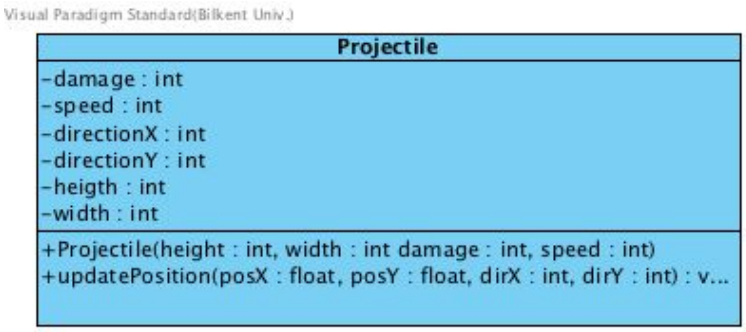
Constructor:

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.

3.3.8 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.

private long startTime: This attribute for determining start time of a projectile.

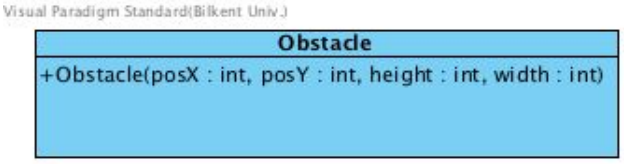
Constructor:

public Projectile (long startTime,int damage, int speed,int typeID, int x, int y, int height, int width, double dirX, double dirY): Constructor with size, damage and speed.

Methods:

public void updatePosition(): This method updates the projectiles position according to its attributes. It basicly multiplies directionX with speed and sets its new location.

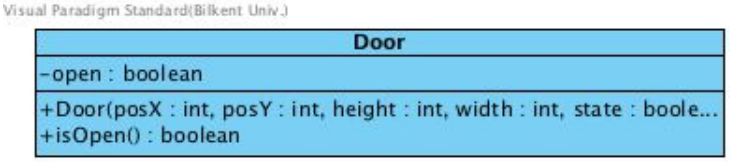
3.3.9 Obstacle Class



Constructor:

public Obstacle (int x, int y, int typeID,int width, int height): Constructor with position and height, width attributes. Position and height - width attributes come from entity class.

3.3.10. Door Class



Attributes:

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

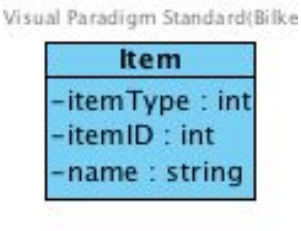
Constructor:

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 entity.

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.

3.3.11 Item Class



Constructor:

Item(int x, int y, int typeID,int width, int height,int itemID): Constructor for creating Items.

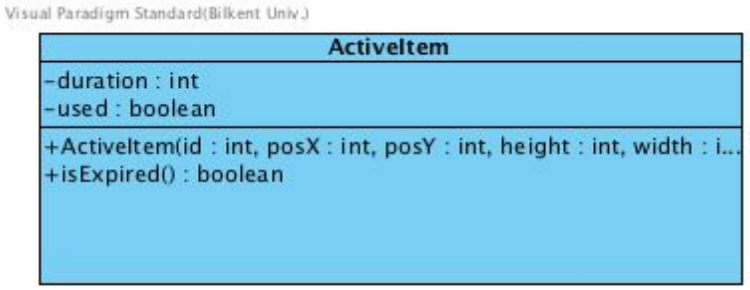
Attributes:

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.

3.3.12 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.

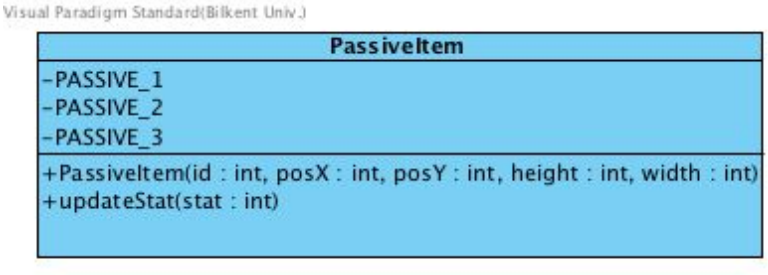
Constructor:

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 Entity.

Methods:

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

3.3.13 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 Entity class.

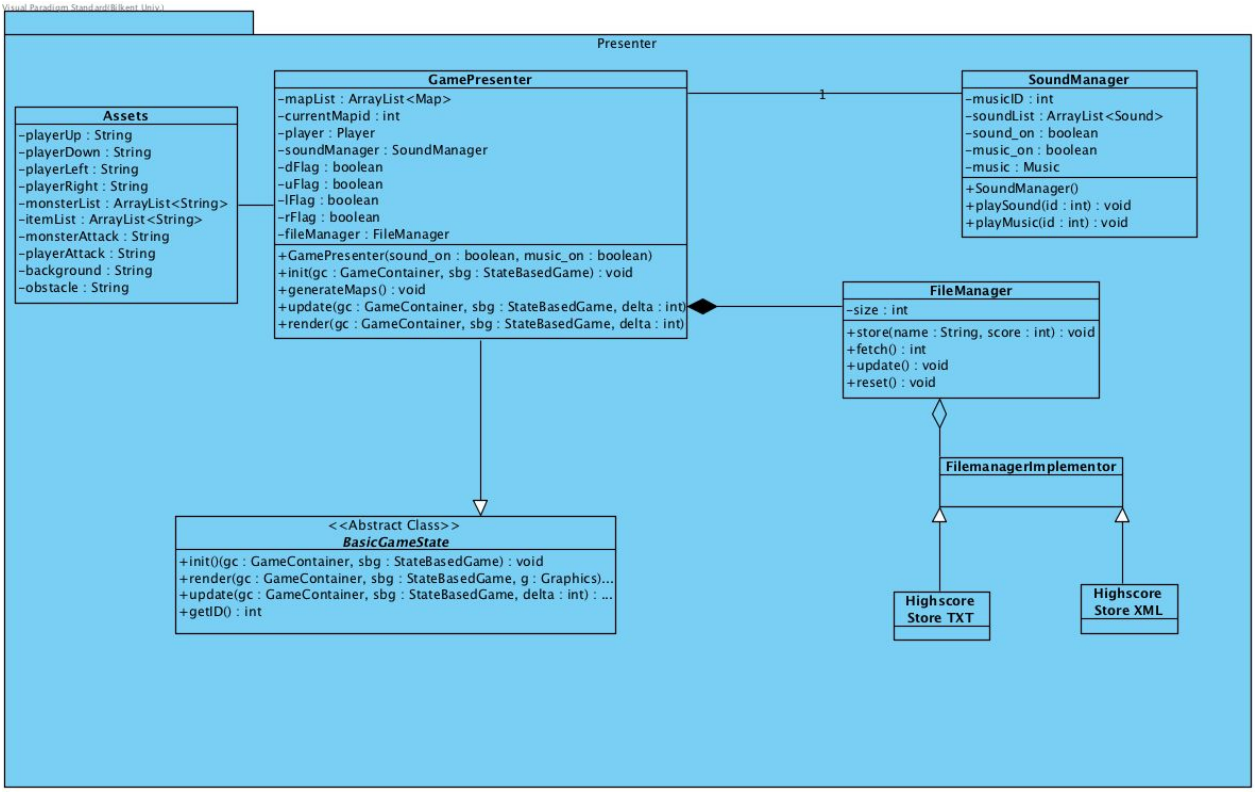
Methods:

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

3.4 Game Presenter Subsystem

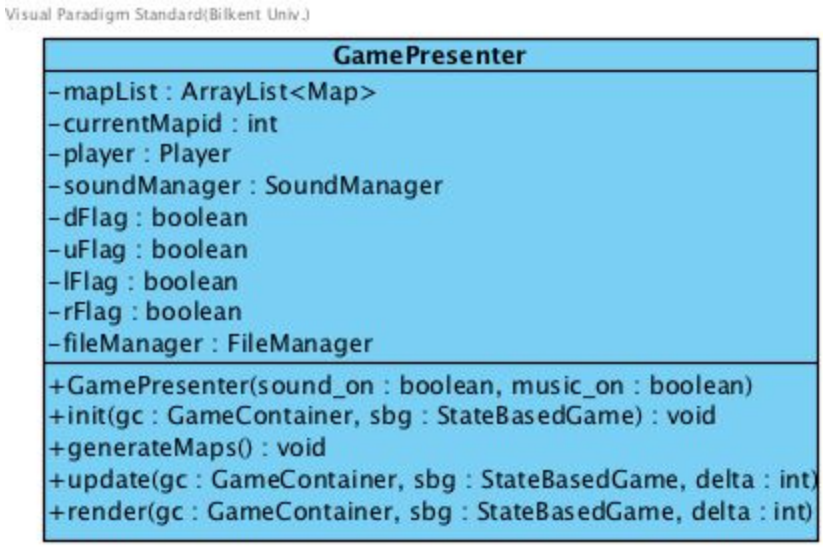
Game Controller subsystem will be the inter-layer in three-tier system which will contain necessary classes to manage user input, game sounds and detect any change in game state. The most important class in this subsystem is Game Manager class has access to all the classes in the subsystem. Game Manager 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.



3.4.1 Game Presenter 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.



Attributes:

private ArrayList<Map> mapList: Holds list of the maps for our game. There are several maps and each map has plenty of rooms.

private int currentMapID: This attribute is for determining current map.

private SoundManager soundManager: This object is for sounds and musics in the game.

private dflag, uflag, lflag, rflag: These attributes are for direction of player. For example if our player is facing left, lflag becomes true and image changes according to it.

Constructor:

GamePresenter(boolean sound_on, boolean music_on): Constructor for Game Presenter. Takes sound on and music on and acts according to it.

Methods:

public void init(GameContainer gc, StateBasedGame sbg): This method is invoked when Game Presenter first initializes. It basicaly draws things which has to be drawn and initialized initially when the game begins..

public void generateMaps(): Generates maps for the game.

public void update(GameContainer gc, StateBasedGame sbg, int delta): Updates the models, in other words logic of the game constantly. For example when player's press "s" button for going down, it updates the characters location attribute.

public void render(GameContainer gc,StateBasedGame sbg, Graphics g): Draws the models into the screen constantly. Projectiles, rooms, monsters etc.

3.4.2 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.

Attributes:

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.

static protected Music music: Music object is for menu music or game music etc.

static protected ArrayList <Sound> soundList: This list containts every sound in game for example player attack or player gets hit or monster hit etc.

static protected final int AN_EVENT: There will be plenty of this attribute. AN_EVENT part will determine whic sound to be played from soundList array. For this report just this attribute is written because it is not certain which sounds will be added.

Constructor:

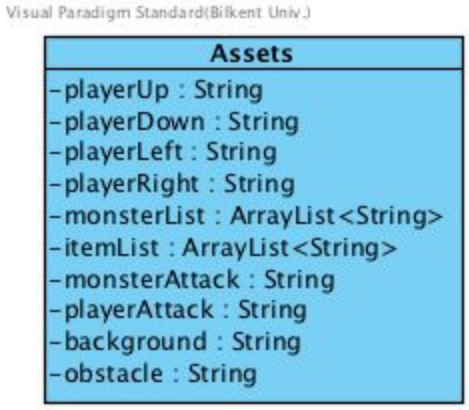
public SoundManager(boolean sound_on, boolean music_on); initializes object, takes sound_on and music_on for enabling or disabling it.

Methods:

public void playSound(int **AN_EVENT**); Takes sound id for example collision sound, projectile sound etc and plays it by using the soundList which holds names of sound. For example 1 for collision, 2 for taking damage etc.

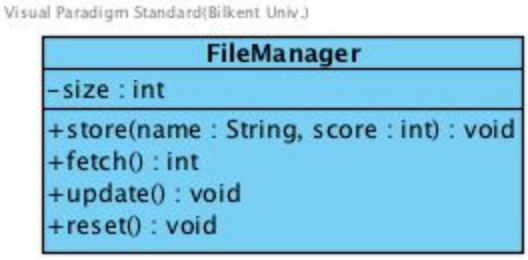
public void playMusic(int musicID); Takes current musicID and plays it.

3.4.3. Assets Class



This class enables us to use images of characters, monsters, projectiles, shortly all entities. This class is static which means that it does not need a instantiation. All attributes are strings which are holding the paths of images used in game. To cut a long story short, Entities use this class for storing images.

3.4.4. FileManager Class



This class is used to store score and corresponding player name in a file format. Bridge Design Pattern is used in order to map the model with the implementation of the class. We have provided abstraction between the real implementation so that we can test different implementations for storing highscore data. This pattern allowed us to decouple *FileManager* class and *FileManagerImplementor* so that former provides high levels of functionality and the latter provides low-level functionality.

Attributes:

size: This attribute is to determine how many scores will be held in file.

Methods:

public void store(String userName, int score): This method stores the data of highscores and name of the user in the file system.

public int fetch(): This method fetches the highscore from the file system.

public void update(): This method is used for updating the size by 1.

public void reset(): This method is used when the itself is rebooted and highscore is set to default value 0.

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. Improvement Summary

- We have added a “design patterns” subsection to the report. We have not used design patterns in our first iteration and this subsection includes the design patterns that we have added in the final iteration. A more detailed explanation about the design patterns that we’ve used is in the “design patterns” part.
- We have changed the architectural style of our project. We decided that our code up until now fits more to the Model-View-Presenter(MVP) architectural style which is an extended and modified version of Model-View-Controller(MVC). The main difference between MVC and MVP is that in MVC, the controller affects the data in model and the view changes its state according to the data in the model while in MVP, there is presenter instead of controller that changes both the data in the model and the view and the other aspects are pretty much the same. A more detailed explanation of MVP in our project is in “subsystem decomposition” part.
- Since we have changed the architectural style of the system, some classes were changed, removed and added when compared to the first iteration.

The classes that were affected by this change are mostly the classes that were in the controller subsystem in the first iteration such as "GameManager" class.

6. 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