Object-Oriented Software Engineering

JAVA BOMBERMAN PROJECT

DESIGN REPORT

Table of Contents

1. OVERVIEW

We are the members of Group #4 in CS 319 course. We are assigned to design and implement an object-oriented programming project. We have decided to work on a Bomberman Game Project with Java. There is a general definition of and brief information about Bomberman in “Introduction” part of this report. After short description of the game, there is an explanation of why Group #4 has chosen to work on Bomberman Game Project. In this report, requirements of the project are coming under two headings: Functional Requirements and Non-functional Requirements. System models of the game will be described in detail. System models section includes domain analysis, use-case analysis, sequence diagrams, state and activity diagrams. After System models section, detailed description of Bomberman game features will be deliberated. Then the report is concluded with evaluation of Game Project Analysis Period.

2. INTRODUCTION

2.1. Purpose of the System

Our system's intention is to provide enjoyable time with a game called Bomberman. Bomberman is an arcade game developed by Hudsonsoft in [1]. In the game that Hudsonsoft developed, he is a robot that works in a bomb factory and tries to escape from there. In our system, the original scenario is a little bit modified. At the beginning of our system, Bomberman tries to get out of a maze and get rid of monsters that are following him. After he passes the maze end escape from the monsters, he proceeds to a new level, with a different place and level of difficulty. At the end, if all levels are passed, Bomberman rescues himself from danger.

2.2. Design Goals

Our main purpose is to develop robust, maintainable, well-designed and reusable software with Object-Oriented analysis and design. We are determined to define and visualize each and every perspective of the system explicitly in order to completely materialize our Object-Oriented approach. Next, we also pay attention to how to diminish the influence and impact of alterations, how to keep the elements of our design understandable, manageable, focused and who is when behavior differs by type.

2.3. Definitions, Acronyms, and Abbreviations

The following is a brief list of acronyms and abbreviations that are used in this document:

GUI: Graphical User Interface

UI: User Interface

JDK: Java Development Kit

J2SE: Java 2 Platform, Standard Edition

JRE: Java Runtime Environment

2.4. References

[1]. Bomberman - http://en.wikipedia.org/wiki/Bomberman

3. PROPOSED SOFTWARE ARCHITECTURE

3.1. Overview

We have introduced the proposed software architecture of our game project in the following subsections. First subsystem decomposition of packages and their interactions are demonstrated. Then software / hardware mapping and persistent data management issues explained. We aimed to design the game in such a way that whole implementation processes would be easier, intuitive and straight forward.

3.2. Subsystem Decomposition

Our decomposed our subsystem of game project into two packages: Game Logic and GUI. GUI Classes are assigned responsibilities such as providing well-designed and easy to understand interaction between user and the game logic. Game Logic Classes are GUI-independent main game components.

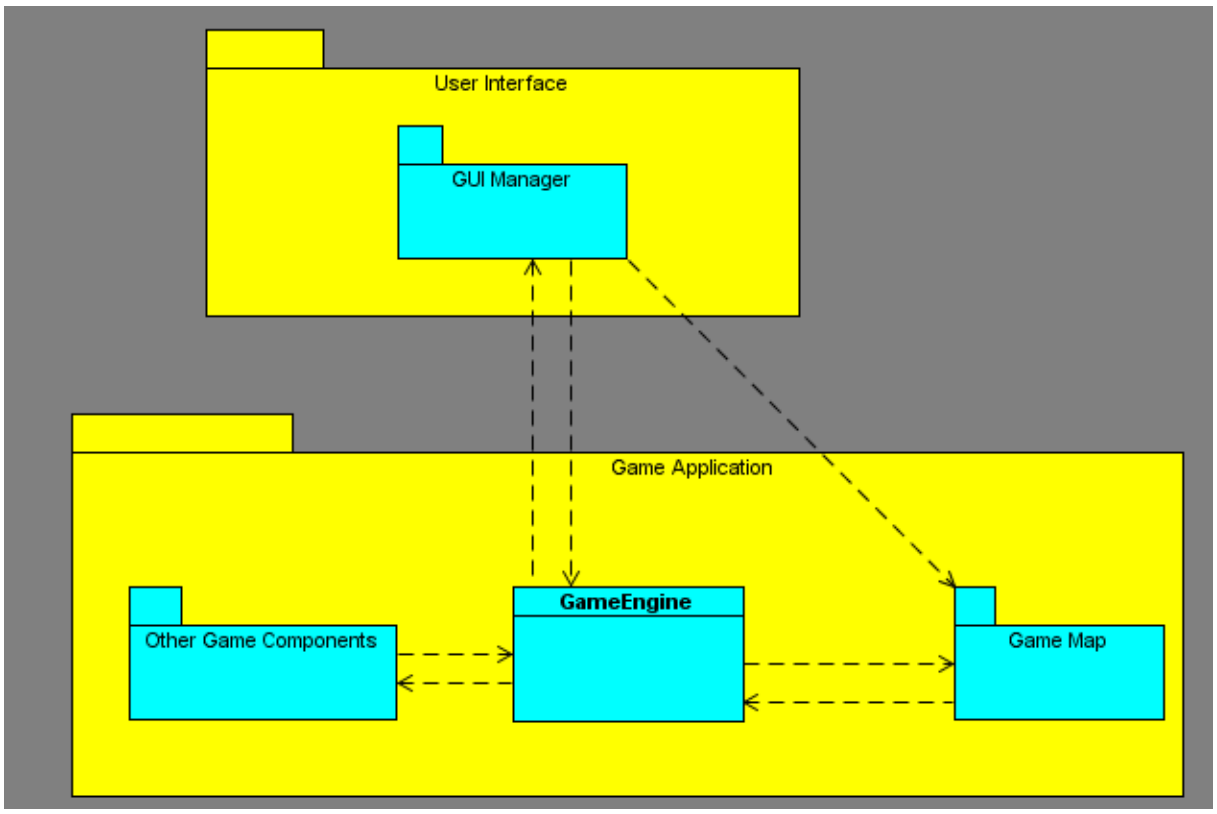

Figure 1: Main Package Relations

In User Interface (UI) package, the game components related to the graphical user interface is considered to be implemented. UI package contains basic GUI screens of the game. GUI Manager Class works in this package. GUI Manager Class manages the game screens and provides communication of GUI components with Game Application layer components accordingly. Game Application package contains game application layer components. *GameEngine* class and Other Game Components package and Game Map package are settled in this package. Other Game Components package contains other objects in the game. Here *GameEngine* class has a prominent role. It is a kind of manager and also connector between *GameMap*, Game Components and GUI Manager. We also provided a link between Game Map and GUI, at first this link seem to be breaking the low coupling idea but since both game map and GUI are relatively completely static between them. GUI will just read the map and depict it on the screen. This causes no harm in terms of object interactions therefore, for performance issues we decided to take out the middle man between game map and GUI.

3.3. Hardware / Software Mapping

The programming language that we will use for implementing the core design of our game project is JAVA. We will use JDK and J2SE Platforms to develop the program and corresponding JRE package will be required to run our Java application on Windows or Linux systems. Bomberman game runs on one PC at a time. (No

multiplayer option and no network connection feature.) Software can be executed with a single executable file with some other files.

3.4. Persistent Data Management

Some files are saved and maintained in order to provide the game some non volatile operations. The game keeps names and scores as high scorers in plain text files in order to display to the users in the Hall of Fame section. Some images and music files are also used at different parts of the game in order to provide better experience on gaming. These files are read from the file system with their specified directions as parameters. Also level data will be kept in a persistent storage (hard drive in this case). At each level of the game there is a different game map. Adjustment information of game maps are also processed from corresponding files in the hard drive.

4. SUBSYSTEM SERVICES

4.1. Game Logic Design

4.1.1. Overview

In this chapter the classes of the project will be explained in detail with a class diagram helping. In this part there are many trivial methods omitted such that simple set and get methods. Those are to be added at implementation stage.

Main relations between components go like; first, game engine initializes with the information flows from GUI components, i.e. name of the player, difficulty of the game, current level and so. Game engine is the administrative component of the logical structure. It talks with game map for information of the surroundings of the player according to players input. Then it decides what to do with respect to this information. For example, if player tries to go to the left, game engine takes the information of corresponding coordinate from game map. If there is a wall game engine prevents player to go that direction or if there is a monster game engine send commends to decrease the lives of the player. One another point is the collision mechanic of the game. We solved this problem by designing the game map as a grid map. Every movement of any dynamic object occurs as discrete grid coordinate movements. Therefore collision problem simplified to the basic coordinate equality checks. If a monsters current coordinate overlaps with players coordinate, player lives decremented. It is that simple. This design choice actually is not about getting away from complexity in return decreased reality. Generic Bomberman game is very suitable for grid map design. In fact original bomberman has this kind of design. Also it is easy to increase the reality by making animations between one grid cell to another smooth, which is not hard with Java's capabilities. Main relations between classes depicted in figure 5.

4.1.2. In Game Activities

Mainly system is in “game loop” which continuously updates the map and saves the activities on it and listens for the player’s inputs. Player can either press one of the directional buttons for moving around or presses a bomb deploy button to setup a bomb to the current coordinate on the map.

There are three parallel processes in this game which are player movements, bomb’s explosions and monster movements. Last process is not in this diagram due to complexity concerns. Monster’s activities are explained in the next diagram. For the bombs when they explode, game checks their explosion range for walls, monsters or players. If there exist a wall, it gets destroyed, if a monster is on the way, its lives decremented or dies, for the last case if player happens to be in the explosion range again he’s treated as monsters, game checks player’s lives and decides if he should continue or drop dead. Additionally for praising the player’s good game play, if player manages to kill three monsters in a row without losing his lives, he is assumed to be in a killing spree which multiplies the score gained afterwards.

On the other hand, movement dynamics goes like this. If player deploys a bomb, its timer starts and waits for its explosion interrupt. Or if player moves around with directional buttons if there is no bomb or wall on the way, “game” lets the player to move that place and series of checks stars. First, game checks the newly arrived location if there is a power up, than if there is a monster on the way. For prior case player yields the power up’s specialty and for the latter case game decrements the player’s lives. All of those processes synchronize before the next iteration of “game loop”. All these activities depicted in the following diagram.

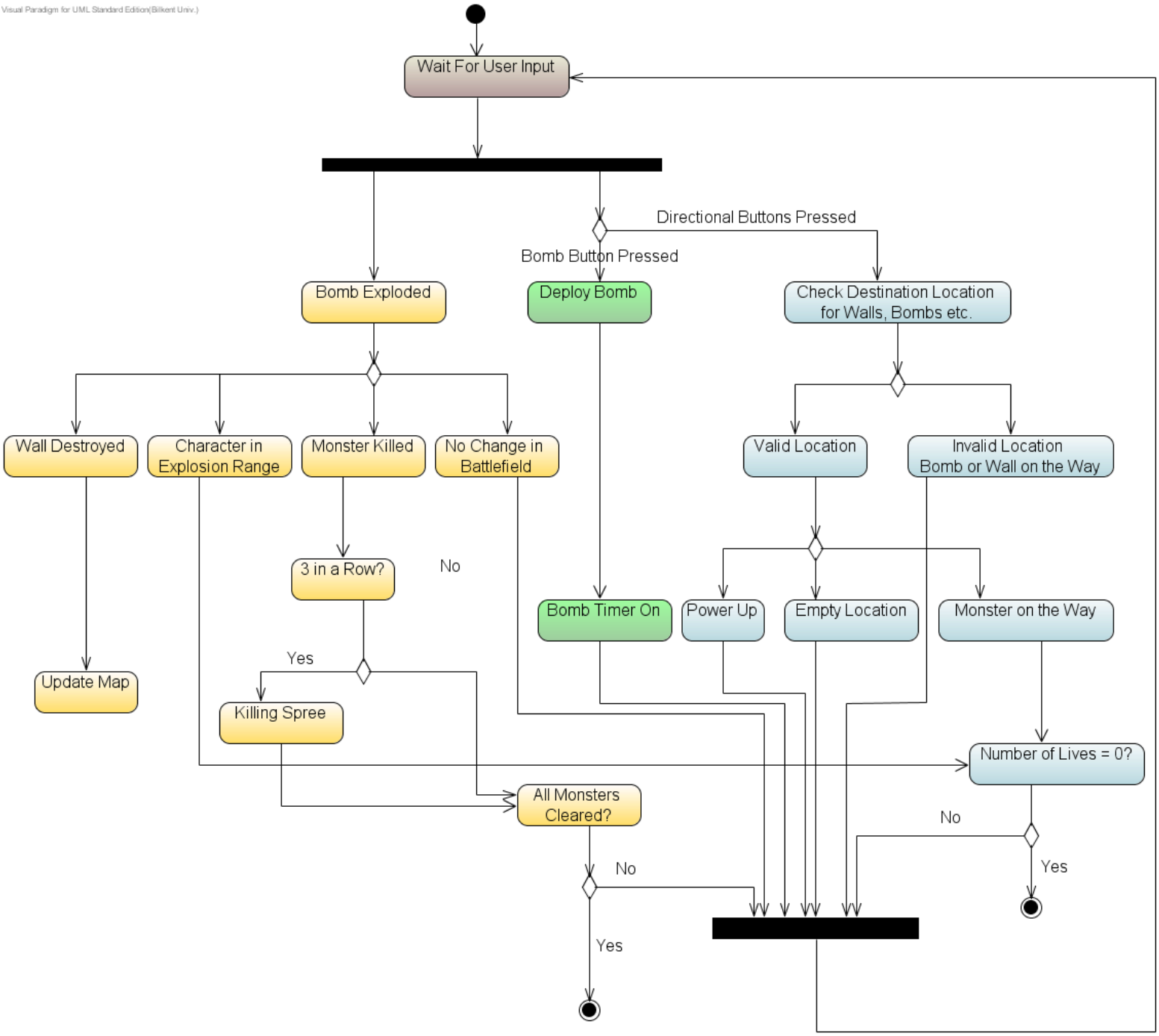

Figure 2: In Game Activity Diagram

4.1.3. Example Scenarios

Scenario 1: Rogerio takes a glance at Hall of Fame section. Then he looks at the high scores.

Firstly, in the game screen which is a GUI Manager component, the player clicks on the high scores button (in our game it's called Hall of Fame button). Then, GUI Manager creates an instance of Hall of Fame screen to display it to the user. Hall of Fame screen gets the related data about high scores via GameEngine class. An instance of GameEngine reads the high scores and scorers from a predefined text file. After that, GameEngine object sends this information to HallOfFameScreen object. It displays high scores via GUI Manager.

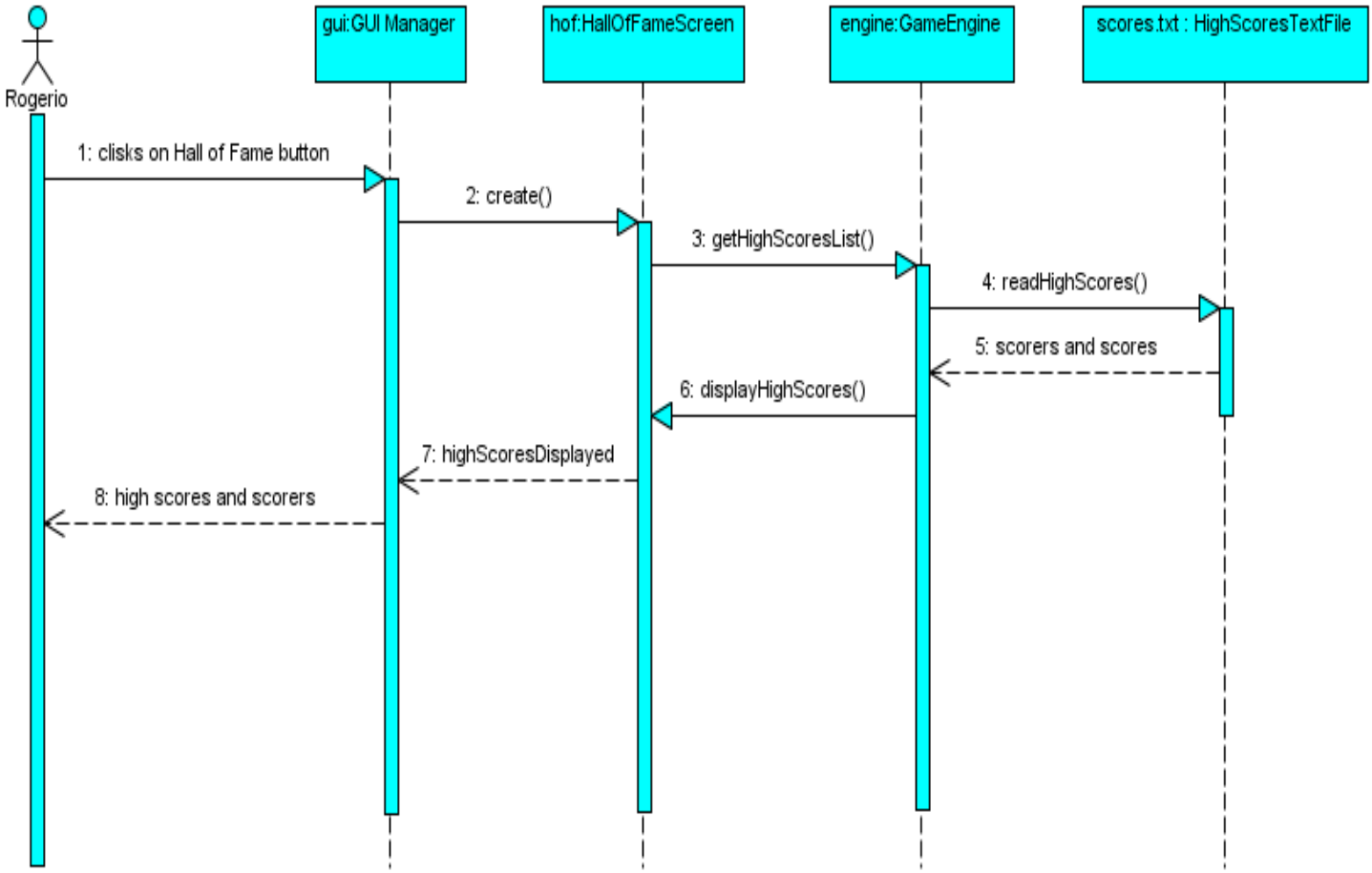

Figure 3: Scenario-1 Sequence Diagram

Scenario 2: During game, Mülayim puts a bomb to a specific location in order to kill a smart monster. However, the monster avoids this bomb. Then the same monster kills Bomberman.

First the player presses space bar button in the game screen. Then GameEngine object is informed about that keyboard event. Player puts a bomb to the location (2,4). Then a Bomb class instance and Timer class instance are created simultaneously. GameEngine instance makes the Timer object start. And also one of the SmartMonster class instance is also directed by GameEngine. SmartMonster object avoids bombs and Bomberman. GameMap object repaints the screen appropriately. When the predefined duration goes down bomb is exploded. Bomb object and Timer object are destroyed. Then the monster touches player and takes away the last life of the player.

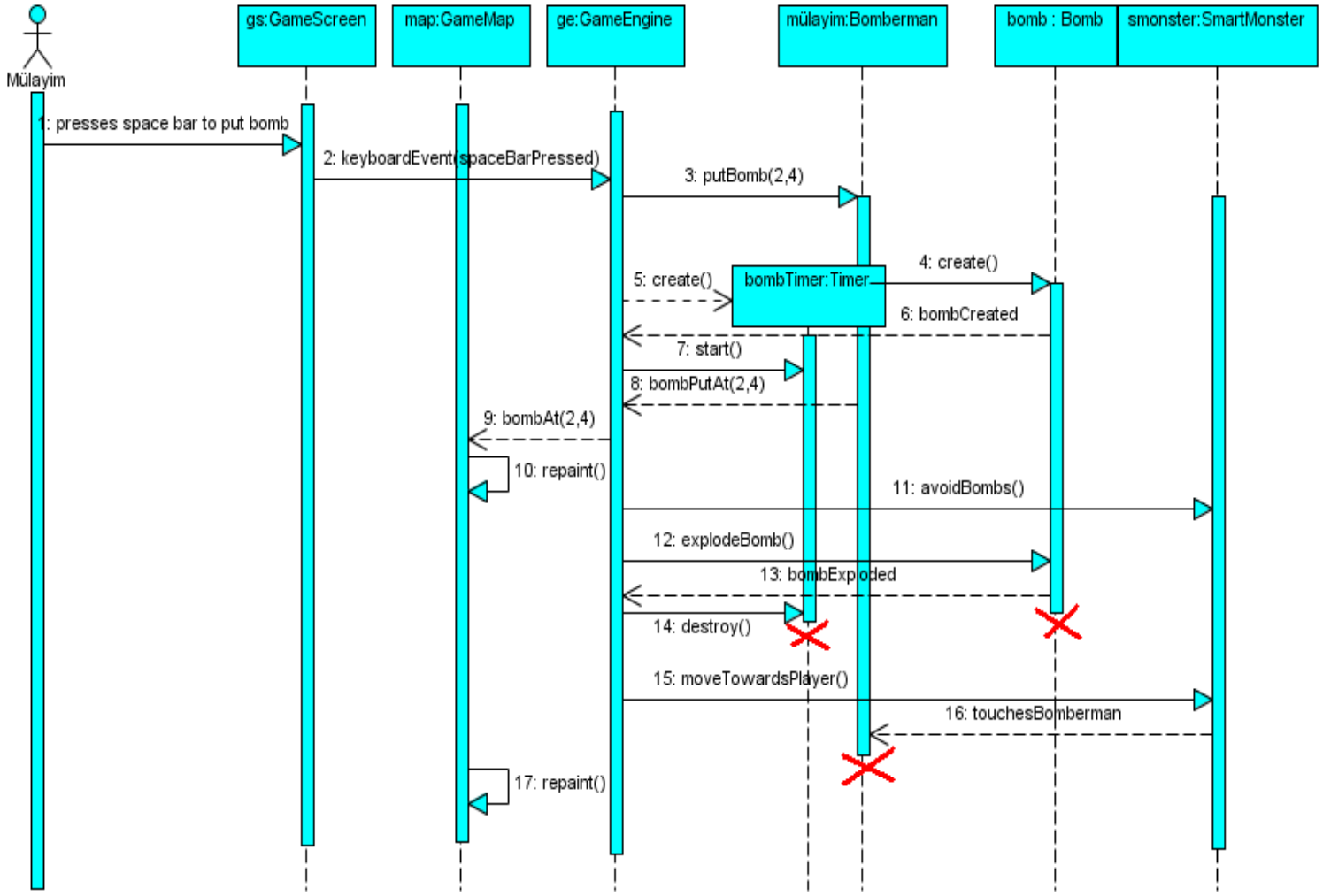

Figure 4: Scenario-2 Sequence Diagram

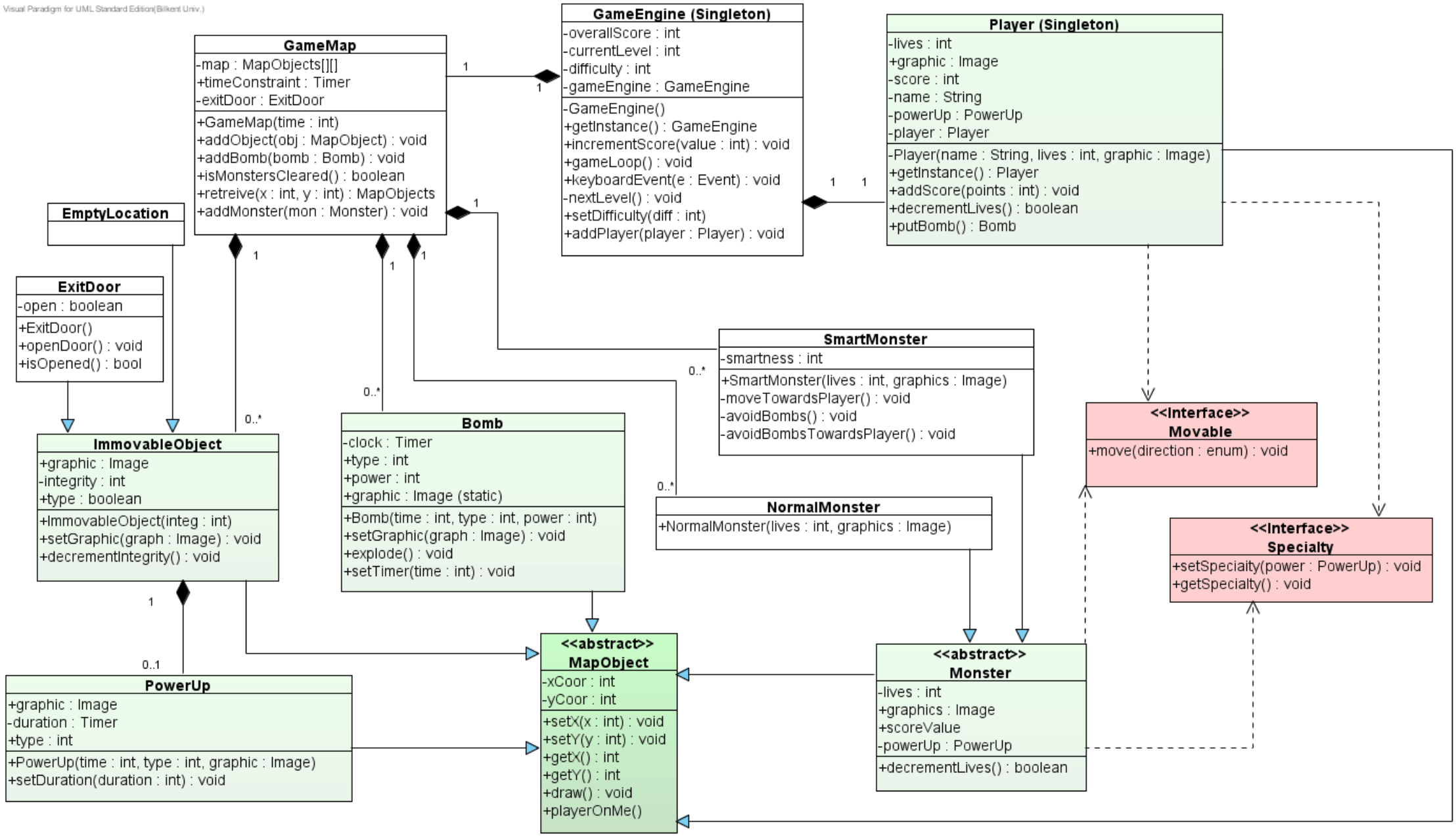

Figure 5: Game Logic Class Diagram

4.1.4. Detailed Class Design

● Game Engine:

Game engine is the main administrative class of the project. It facilitates the operations between classes and provides some of the algorithms for the game. Game engine is a singleton class because it is designed to be unique for any game. It requires a player and it will not change until exit.

Properties:

"*player*" is the person who currently plays the game. It means player object created or read from file as loading procedure after GameEngine instance is created. Player object is not to be destroyed until game is executed.

"gameMap" holds the level information for the currently played level. This means when a level is passed, one gameMap object is destroyed and next level's game map created and assigned to currently active game engine.

"overallScore" is the sum of the scores earned from previous levels. Since player object merely interacts with the currently active gameMap object, we decided to let player's score counter, count the earned point from current level. Also since one gameEngine object represents one user's whole game it is convenient to put overall score to game engine.

"currentLevel" is just an integer value to represent currently played level. It is going to be used for loading correct levels from persistent storage.

"difficulty" is again not in the player object but in game engine. Because game engine is the one represents the current game session and it has the difficulty property to pass on monsters and so.

"gameEngine" is this class' one and only instance as GameEngine designed to be a singleton class.

Methods:

"incrementScore" is used for adding the earned score from previous level to the overall score.

"gameLoop" is the main logical loop of the game. As long as level continues this loop does not end. This loop calls the necessary functions of the various objects to manage the game properly.

"keyboardEvent" is the main function that facilitates the input to the application from keyboard.

"nextLevel" function will be called after a level passed successfully. It does the housekeeping of the level ending and starting a new level. It will be called by gameloop therefore it is only gameloop method's responsibility to call it accordingly. Hence we set this method to be private.

"getInstance" is used for creating the first instance if the class and preventing the second or more instances.

"setDifficulty" is used for determining the difficulty for the game. It will be called just once for a game.

"addPlayer" will be called once and only after the creation of the instance of game engine.

Also for this class it is hard to imagine the size of the code without implementation, therefore we may extract some other operations from game loop as another distinct method. This way we would increase the modularity of the design.

● Game Map

Game Map is another big class of our design. It has all the information of a level's map. It is responsible of providing information of all the map objects currently on the map and providing collision information between map objects. It can be built with add methods of which their parameters comes from map file saved on a persistent storage.

Properties:

"map" is the main two dimensional array of the map objects. Everything on the game map grid is a map object. We decided to let this grid's dimensions to be static means size of the levels will be constant throughout the game. Size of this array will vary between 20*20 to 20*40.

"timeConstraint" is simply responsible of time constraint of the level. As we have stated in analysis report our game has a time limitation for every level which is variable with respect to difficulty.

"bombs" is the list of bomb objects which holds all bomb information in a level. Since player actually cannot set two bombs at the same time but there are power ups which provide this property, nonetheless for this list it is enough to use a small sized array because even with power ups user will not be able to put 5 or more bombs at the same time.

"staticObjects" is the list of immovable objects in the map. Those objects are namely walls, trees, bushes or so. Immovable objects will be explained in detail later. As a matter of fact this array is the most fundamental property for the design of a level. This data will be encoded in a "dat" file as data for a level.

"monsters" is the list of currently active monsters on the map. From the perspective of the GameMap object monsters are static in a sense that their motion will be handled by game engine and game map object will only be aware of the new locations of the monsters.

"exitDoor" is one another immovable object but it is not in the staticObjects list, because its responsibility diverges from other immovable objects. It will be opened after all of the monsters cleared from stage. And when player enters this door level will end.

Methods:

"addObject" is used for adding any map object to the game map.

"addBomb" will be called when player sets up a bomb. This method adds the bomb to the bombs list and to the map array.

"isMonstersCleared" is used for querying the clearance of the level. When this method returns true, it means exit door will be opened for ending the level.

"retreive" is used for peeking to the objects in the map array at the corresponding coordinates.

"addMonster" will be used by game engine for the purpose of initializing the level for the game. Monsters will be scattered to the place both by randomly and sometimes by using the level data from persistent storage.

● Movable <<interface>>

Movable is an interface which adds the functionality of move to the classes which implements movable. The only method in it is the signature of move method. Both player and monster classes implements this interface because

they are the only movable objects in the game map. This method takes an enum type named direction as “left, right, up, down”. Despite the fact that, move method is empty because it resides in an interface class, it supposedly should move the movable object in its coordinate system accordingly to the direction parameter.

● Specialty<<interface>>

Specialty is an interface class, its purpose is creating an specialty interface between objects which their classes implements this interface and other object which associates with.

It has two method signs, “setSpecialty” and “getSpecalty” they provide the ability to objects to have a powerUp in their service. Both player and monster classes implement this interface, because player can grant a power up by walking over it. And monsters have a specialty of a power up for all of their lifetime.

● MapObject <<abstract>>

This is the base class for all the objects exist on the game map grid. We designed this class as abstract because it is simply a template for map grid objects. This class simply provides coordinate values as integers and their get and set methods. Also this class provides *playerOnMe* method to determine the activity of the system if player goes into this coordinate. For example if player pressed left button players left cell coordinate will be sent to the *gameMap*, than *gameMap* will call the correspondong map object’s *playerOnMe* method if this object is a wall, it just won’t let player coordinates to be changed or if this is a monster player’s lives will be decremented and coordinate will be updated. By this design we take advantage of the polymorphism. Aldo the *draw* method here provides drawing method to all map objects for to be called by GUI when refreshing the screen.

● Player

Player is another singleton class in this design. It will be created at the start of a new game or loading an old game. After that it will be passed to game engine and game will start. Player is responsible from holding the information of number of lives it has and current score of the level. Every time player passes a level current score will be added to the main score held in the game manager and next level’s score will be start from zero. Also since player can move and have a special ability via power ups, it implements Movable and Specialty interfaces. Also this class inherits from MapObject class.

Properties:

"lives" is simply number of lives player has. If it reaches zero game is over and sum of the score will be passed to the GUI.

"graphic" is the image file of the visual representation of player. This will be changeable at the settings menu.

"score" is as explained before is the partial score of the game. It holds amount of score earned in current level.

"name" will be provided at the start of the game it will be constant throughout the game.

"powerUp" is a PowerUp object which represents player's current power or it can be empty as player's initial state.

"player" is this class' one and only instance as Player designed to be a singleton class.

Methods:

"getInstance" is used for creating the first instance if the class and preventing the second or more instances.

"addScore" is used for adding the earned score to the partial score of the player. For example if player takes a score boost power up, this method will be called for incrementing the score of the player.

"decrementLives" method will be called if player loses a life.

"putBomb" method is used for setting up a bomb to the coordinate in which player resides. This method by default will be initiated with "space" key on the keyboard. When this method called this player object will create a Bomb object corresponding to the power up he has. This powered Bomb object will be returned to the *GameEngine,* than game engine will pass it to the *GameMap* which will be added to the map array and timer of the bomb will be started.

● Monster <<abstract>>

This class is the base class for two types of monsters as NormalMonster and SmartMonster. This class is designed as abstract because it is a template for monsters.

Properties:

"lives" holds the number of lives of the monster. If it reaches zero monster dies and cleaned up from game map also its score will be passed to the players score.

"graphic" is the image file of the visual representation of monster.

"scoreValue" is the amount of points which will be added to the players scoreboard when killed.

"powerUp" is the specialty of the monster, it will be assigned at the start of the life time of the monster. Monsters cannot grant different powers in their lifetime.

Methods:

"decrementLives" method will be called if monster will be in an explosion zone of a bomb. Game engine is responsible for calling this method when necessary.

● NormalMonster

This class represents monsters without any artificial intelligence. This kind of monsters will move on the map either on a pre determined circuit or randomly. This class inherits from Monster class. Movement of this monsters will be managed by game engine by its move function (inherited from Movable interface).

Properties:

"smartness" is the value determines the monsters behavior as three different characteristics. If this value is "1" all moves of the monster will be determined by moveTowardsPlayer method. If this value is "2" all moves of the monster will be determined by avoidBombs method. If value is "3", avoidBombsTowardsPlayer method will calculate all moves. Otherwise if value is "4" all three methods will calculate monster's move in turns.

Methods:

"moveTowardsPlayer" method will calculate an adjacent grid cell in game map which is closer to the player. This method will be called with respect to smartness value of the monster.

"avoidBombs" method will calculate an adjacent grid cell in game map which is further from any bombs. This method will be called with respect to smartness value of the monster.

"avoidBombsTowardsPlayer" method will calculate an adjacent grid cell in game map which is both further from any bombs and closer to the player. This method will be called with respect to smartness value of the monster.

● SmartMonster

This kind of monsters has artificial intelligence as their movements are not random or predetermined. These monsters move in three different ways. They can move towards to the player, they can avoid bombs or they can do both at the same time. This class inherits from Monster class.

● Bomb

Bomb is the class of bombs released by player onto the game map. They have a timer for their explosion; timer's value can be change via power ups which player grant. Also their explosion zone will be variable with respect to the player's power up. Additionally this class inherits from MapObject class.

Properties:

"clock" is the Timer instance for the explosion time's of the bombs.

"duration" is exact explosion times which is variable with respect to player's power up. This variable will be passed to the clock property to set the Timer instance.

"power" is bomb's explosion range. This property varies with respect to player's power up. This property will be used by game engine when calculating objects which would be affected by the explosion.

"graphic" is an image file as the visual representation of the bombs in the game map. This property is static because there is only one type of bomb graphic therefore by this way we eliminate duplicate property redundincy.

Methods:

"setGraphic" method will be provided with an image file at the start of the game and it will set the graphics of the bombs objects.

"explode" method initiates an explosion which will be passed to the game engine. After game engine decides the changes in the game map, this method will be called by Timer object of the bomb.

"setTimer" is used for initializing the Timer of the bomb. Its value will be provided by game engine dynamically. As some power ups change the explosion time of the bombs.

● PowerUp

PowerUp is the class of special abilities of the monsters and player's. Those special abilities assigned to the object via this class's instances. It has many varieties indicated by its type property. Also this class inherits from MapObject class.

Properties:

"graphic" is an image file as the visual representation of the power ups in the game map.

"duration" is the active Timer which starts to count backwards as soon as player takes this power up. When this duration time depletes, player has no longer special abilities.

"type" is the representation of different power ups. This integer value will be used in switch cases at the player and monster objects.

Methods:

"setDuration" is used for setting the active time for usage of this power up.

● ImmovableObject

This is the class of static objects in the game map. Walls, trees, bushes or such objects on the scene are not in motion but destructible. Also this class inherits from MapObject class.

Properties:

"graphic" is an image file as the visual representation of the immovable objects in the game map.

"integrity" is the power of the immovable object in the scene. It decreases every time it is in an explosion zone of a bomb. If this value reaches zero, it means object is destroyed from thr game map and now its place is reachable by monsters and player.

"type" determines if the object is indestructible (invincible to the bombs) or not.

"power" is the power up reside in the immovable object. When this object destroyed this power up will take its place.

Methods:

"setGraphic" method will be provided with an image file at the start of the game and it will set the graphics of the object.

"decrementIntegrity" method will decrease the integrity value of the object when called. Game engine is responsible to call this function when a bomb explodes near it.

● ExitDoor

This class represents the exit door which lets the player finish the level. It is going to be opened when all the monsters cleared from map. Game engine is responsible from managing exit door. This class inherits from ImmovableObject class because essentially this door is an immovable object to. It is going to be indestructible by its type setting (comes from parent class).

Properties:

"open" is the state of the door. If this property true door is open. In other words all the monsters are cleared.

Methods:

"openDoor" method sets the open property as true. In other words opens the door.

"isOpened" used for determining if the door currently opened or not.

● EmptyLocation

This class represents movable lacations on the map. This locations can have a power up on them too. For instance if an unmovable object destroyed by bomb explosions, and if this immovable object has a power up in it, after explosion there will be an empty location object at the same coordinate with a power up on it. Player can just walk on it and grant the power.

4.2. GUI Design

4.2.1. Overview

In this GUI Design Part, the menu screens and the relationship between them will be explained with the help of class diagrams, activity diagrams and screenshots of these menus which will be almost the same for the implementation progress.

First of all, our program will use Java's Swing library as GUI components. (javax.swing library) We have designed some screenshots for the requirements of a bomberman game. Then we found out the action flow of the game such that "which screen appears when a certain event happens" or "what needs to be done in the game logic when a button is pressed", etc. According to them, we sketched activity diagrams in Visual Paradigm for the total activities done in the GUI part. Besides this, we draw the class diagram for GUI components of our project. Since they mostly consist of swing library components like JButton, JLabel, etc. there are not very different types of attributes, they are generally similar, just different in the components types and amount.

4.2.2. GUI Components

Initially, the main screen scene in the game as a GUI component is the Main Menu Panel. In this panel, there is a label like a header which involves our project's name "Bombacı Mülaim". Moreover, there is a panel which will contain an image about bomberman that we haven't decided yet. There are 5 buttons to start the game and for the transition among other menu panels.

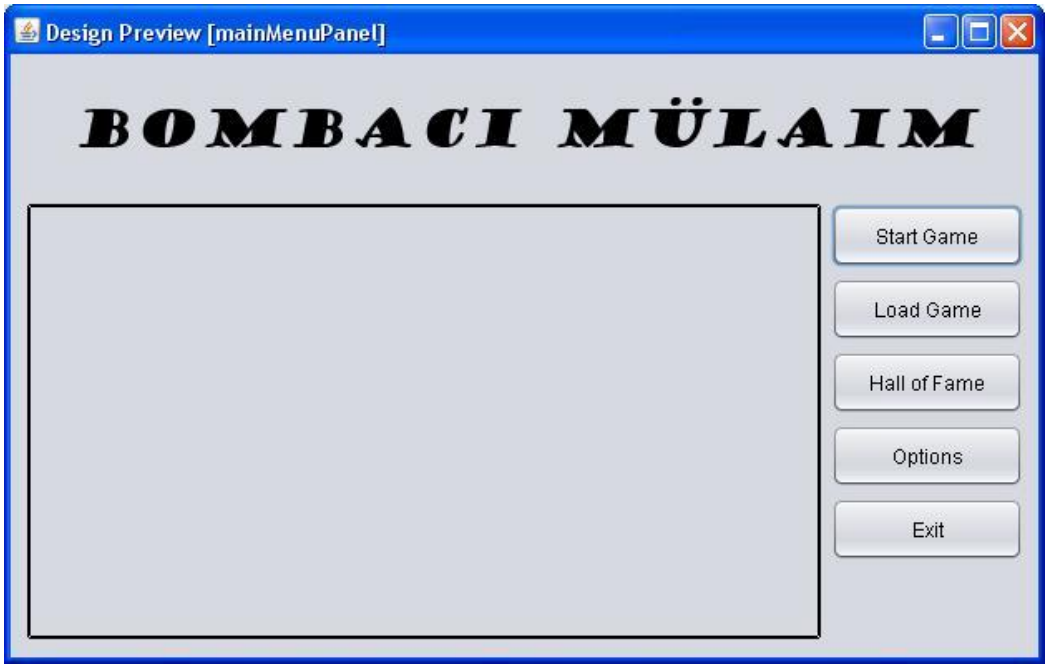

Figure 6: Main Menu Panel

In the Load Game Menu, there are buttons which are for the saved games before.

Figure 7: Load Game Menu

In the Settings Menu, there are choices for the game type, for the volume adjustments of background music and sound effects, for the character and difficulty of the game. At the bottom, the user can configure the keys used in the game.

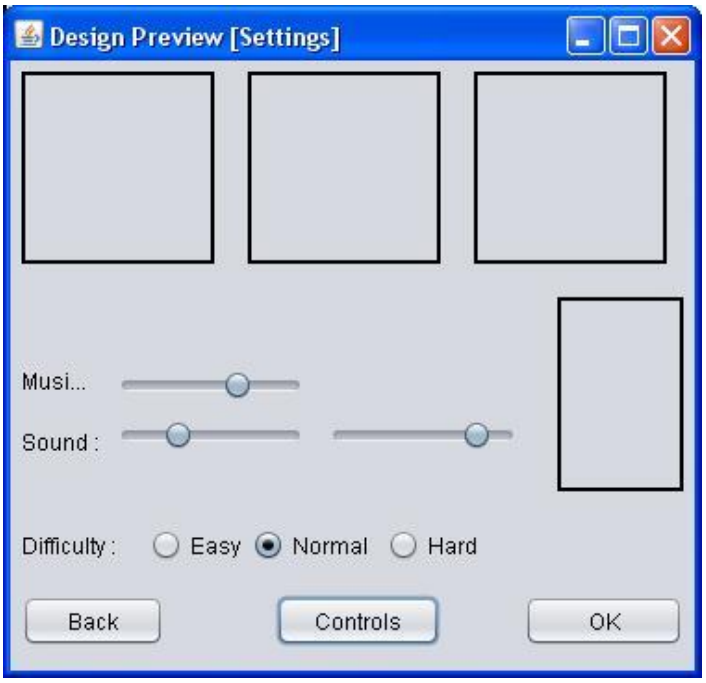

Figure 8: Settings Menu Panel

In the Controls Menu, there are 5 buttons for each key which will be used in the game and there is a button to reset those keys to the default value which the most common keys are used in games.

Figure 9: Game Controls Menu

In the Hall of Fame Menu, the 10-most points done in the game will be displayed.

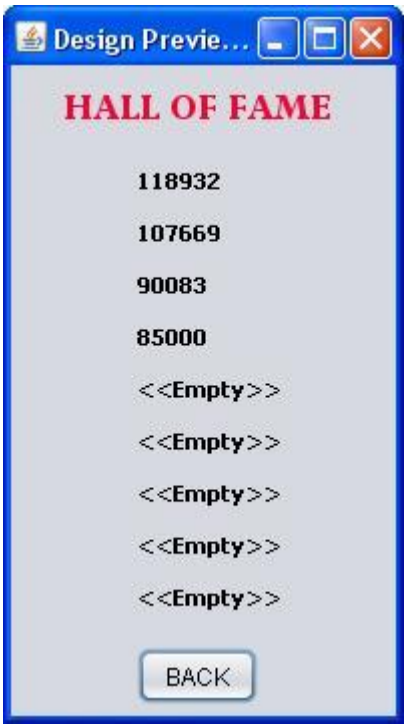

Figure 10: Hall of Fame Menu

In addition to the Main Menu Panel Screen, there is a Pause Menu Panel that the user can adjust some settings during the game and save the current game or load a saved game.

Figure 11: Pause (In Game) Menu

In the Save Game Menu, the user can save the current game.

Figure 12: Save Game Menu

In the Pause Menu, the Options Menu Panel is a bit different than the one displayed through the Main Menu. The configuration of the controls and adjustments of the sound effects are in the same menu.

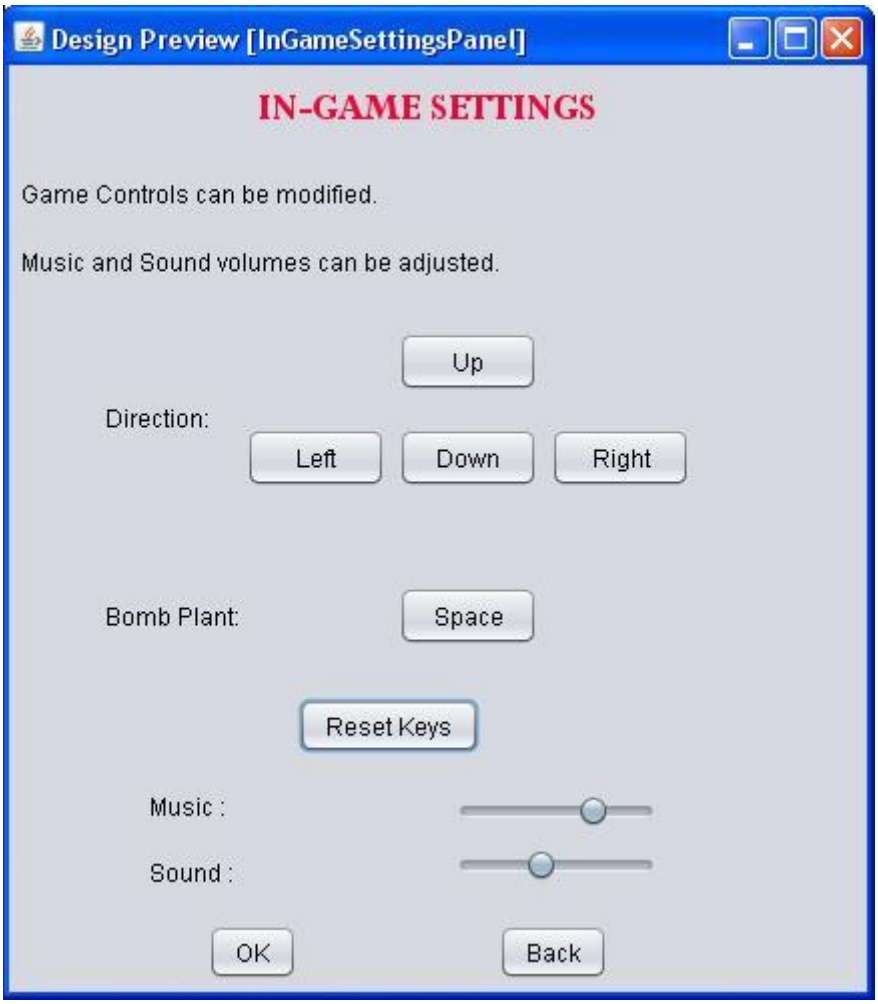

Figure 13: In Game Options Menu

Other than these menus, there are 3 warning and information messages.

If the user wants to exit the game through the Pause Menu, the program wants the user to confirm the exit action.

Figure 14: Exit Warning Message

If the user saves the game successfully, the game informs the player that the saving action is done successfully.

Figure 15: Successful Save Message

If the user wants to save the game into an already existing saved game, the game warns the user and asks if the user wants to overwrite to the current saved game.

Figure 16: Overwrite Save Message

4.2.3. GUI Class Design

In the project, we decided to use the swing library of Java (javax.swing) as GUI components because there are more and usable components and they are much better in the usage. We have JPanels as menu screens and for the transition among panels, we used JButtons.

Since we use many Java GUI components, there are a lot of instances of them in every menu and if we had used them in the class diagram, there were going to be much more and much confusing arrows among classes, so we didn't use the Java Swing class and the components in the class diagram.

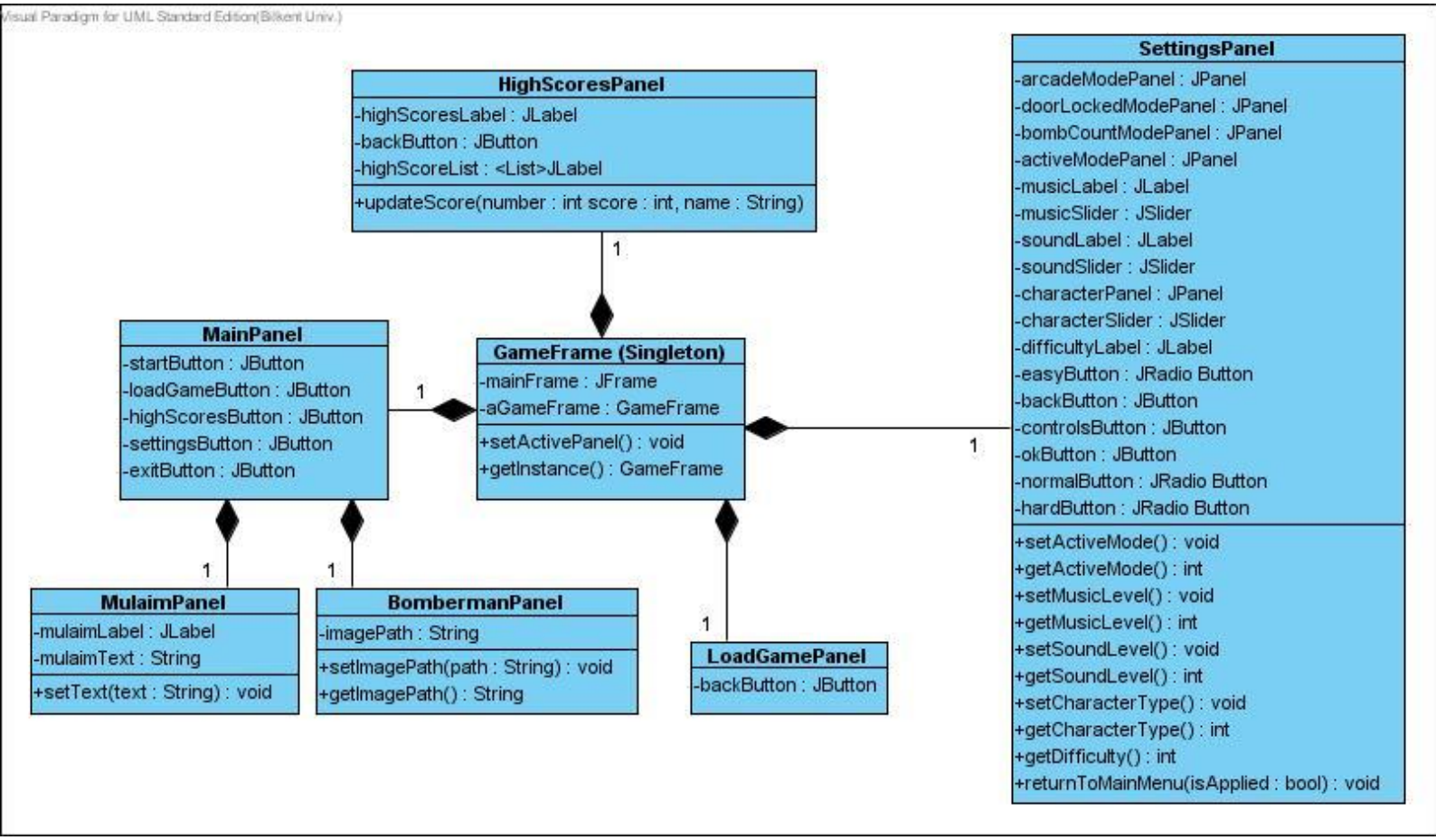

Figure 17: GUI Class Diagram

4.2.4. GUI Activity Flow

GUI activities start in the main menu which will be displayed at the start of the game. If the user clicks the “start game” button, the game starts. If the user clicks to “Load Game” button, the load game menu appears.

The user chooses a saved game from the buttons. If there is no game saved to the location for example to “Game 10”, that button will be deactivate and user is not able to click to it. If he/she does not want to load game, after clicking to the “Back Button” the main menu screen appears.

In the main menu, after the user clicks to the “Options Button”, he/she sees the option menu where there are the settings for the game such as game mode, volume and difficulty adjustments, character choosing and game control settings. If the user changes them, the changes are applied and the options screen still visible. To set the game controls, he/she needs to click the “Controls Button” and set them by clicking the desired button and click the button he/she wants to use. After clicking the save button, the current settings will be saved. If he/she clicks to “Reset Button”, the default game control values are applied.

If the user clicks to the “High Scores Button”, the 10 most points are displayed in the screen.

To exit the game, “Exit Button” must be clicked.

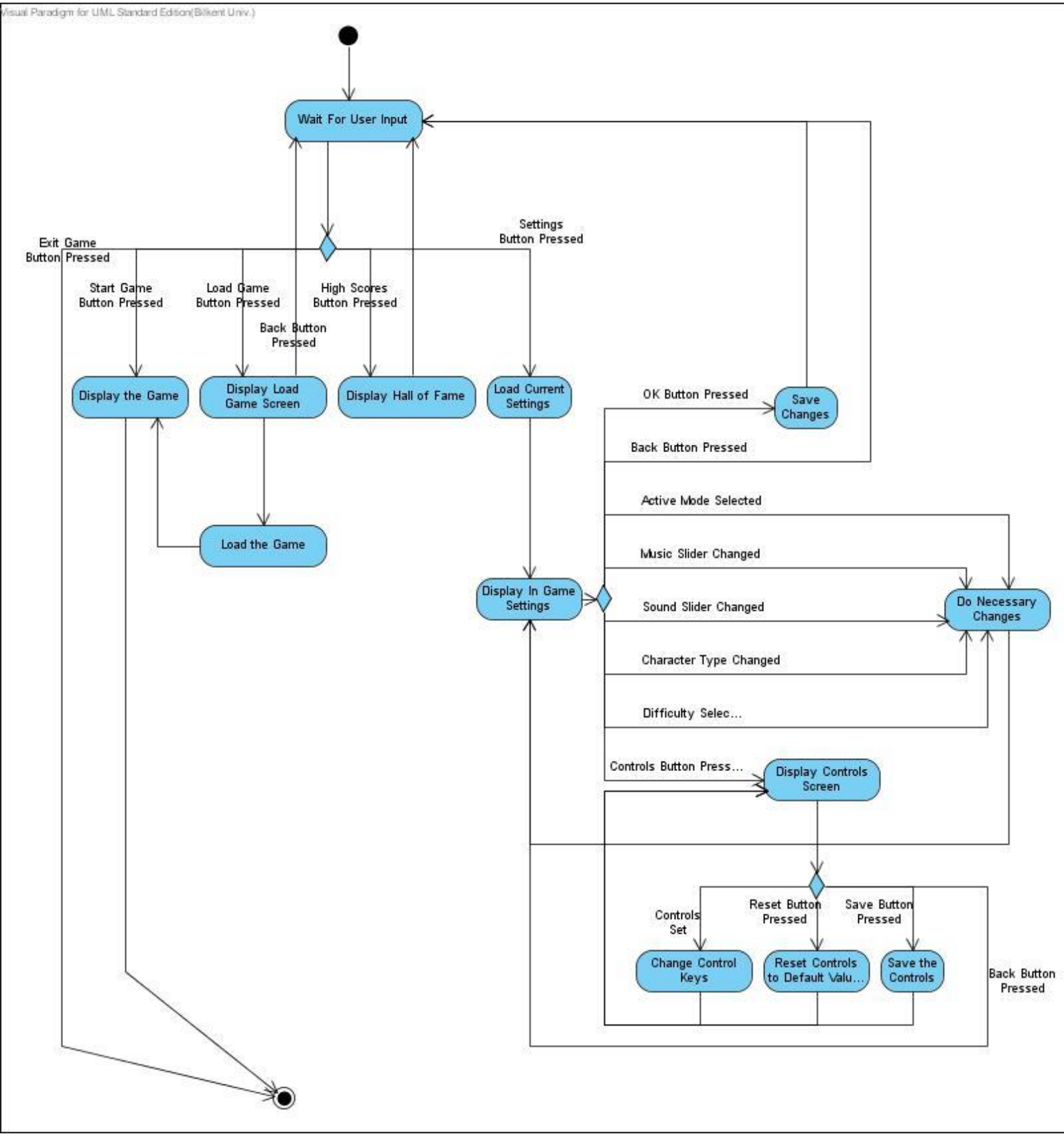

Figure 18: Main Menu Activity Diagram

Additional to the Main Menu, there is a Pause Menu which is for saving, loading and the settings that needs to be done during the game. The Pause Menu appears when the user clicks to “Esc” key on the keyboard. In the Pause Menu, there are 5 buttons for the user; “Continue”, “Save Game”, “Load Game”, “Options” and “Exit Game” buttons. To continue the game, “Continue” button must be clicked. If the user clicks to “Save Game” button, the Save Game Menu appears.

In the Save Game Menu, there are 10 possible locations to save the game. He/She clicks to one of them. If there is an already saved game in that location, a warning message screen opens and asks to the user whether he/she wants to continue or abort saving operation. If he/she continues, the game is saved to that location and an information screen appears that the operation was successful. If the user chooses an empty location to save, the information screen appears and the saving operation is done. After clicking the “Back Button”, the pause menu appears.

In the Pause Menu, the loading sequence is the same as in the main menu.

In the Options Menu in the Pause Menu, the controls and other settings which are in the options menu in the main menu are in the same screen. The activities are the same as in the main menu.

If the user clicks to the “Exit Game” button, a warning message appears in the screen that requires the confirmation of the user for exiting from the game.

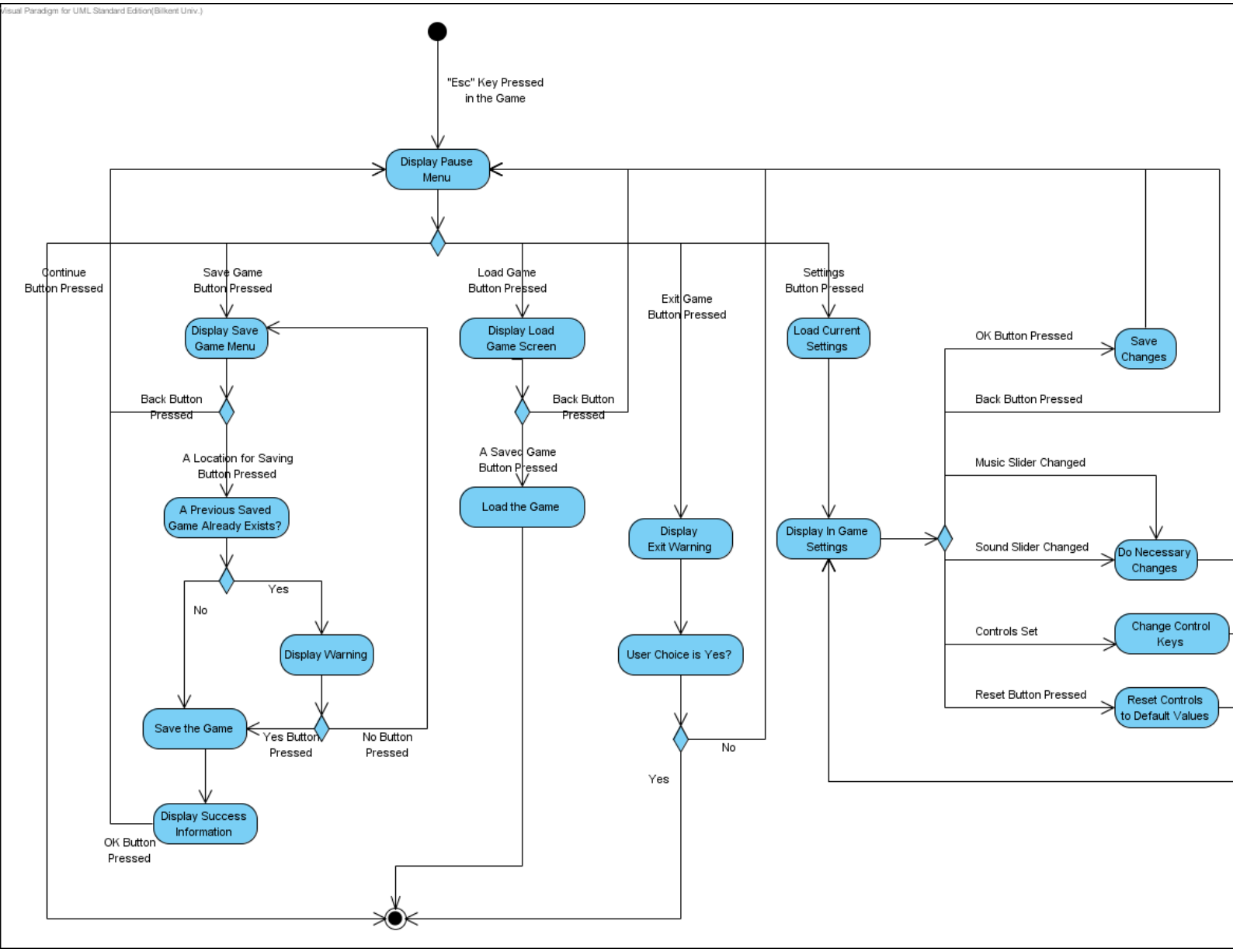

Figure 19: Pause Menu Activity Diagram

4.3. Sound System Design

There are mainly 2 kinds of sound in the game; “Sound Effects” and “Background Music”. We will be using java’s built in capabilities for the sounds.

4.3.1. Sound Effects

There will be some sound effects during the game such as bomb explosions and monster killings. Those are to be initiated just after a monster killed and a bomb exploded. As stated before java has very useful built in capabilities to facilitate the sound effects.

4.3.2. Music

There will be a background music that will play in the menu screens.