



Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering Project

CS 319 Project: Split-Field

Analysis Report

Ali İlteriş TABAK, Efe TUNÇGENÇ, Gizem ÇAYLAK

Course Instructor: Hüseyin Özgür TAN

Progress

Oct 28, 2015

This report is submitted to the Github in partial fulfillment of the requirements of the Object Oriented Software Engineering Project, course CS319.

Table of Contents

1.	Introduction	5
2.	Requirement Analysis	6
2.1.	Overview	6
2.1.1.	Control.....	6
2.1.2.	Line-Rider	6
2.1.3.	Enemies	7
2.1.4.	Power-ups.....	7
2.2.	Functional Requirements.....	7
2.2.1.	New Game	8
2.2.2.	Play Game	8
2.2.3.	Load Game	10
2.2.4.	Pause Game	10
2.2.5.	Save Game.....	10
2.2.6.	Exit Game.....	10
2.2.7.	Show High Scores	10
2.2.8.	Show Help	11
2.3.	Non-Functional Requirements.....	11
2.4.	Constraints.....	12
2.5.	Scenarios	12
2.6.	Use Case Models	14
2.6.1.	Use Case Descriptions.....	15
2.7.	User Interface	23
2.7.1.	Navigational Path	23
2.7.2.	Game Menu Mock-ups	24
2.7.3.	In Game Graphics.....	29
2.7.4.	Characters.....	29
3.	Analysis	33
3.1.	Object Model.....	33
3.1.1.	Domain Lexicon	33
3.1.2.	Class Diagram	34
3.2.	Dynamic Models	39
3.2.1.	State Chart	39

3.2.2.	Sequence Diagram.....	40
3.2.2.1.	New Game	40
3.2.2.2.	Load Game	43
3.2.2.3.	Pause & Save Game	46
3.2.3.	Activity Diagram	49
4.	Conclusion.....	50

Figure Table

Figure 1: Playable character “Line-Rider” [2]	7
Figure 2 Some of major enemies from different levels [2]	7
Figure 3. Use Case Diagram	14
Figure 4. The navigational paths between the game menus	23
Figure 5. The mock-up of the main menu	24
Figure 6. The mock-up of the high scores menu.	25
Figure 7 The mock-up of the help menu	26
Figure 8 The mock-up of the play game menu.....	27
Figure 9 The mock-up of the play game menu.....	28
Figure 10. The mock-up of the game field	29
Figure 11. LineRider.....	29
Figure 12. Level 1 Major Enemy	30
Figure 13 Level 1 Minor Enemy	30
Figure 14. Level 2 Major Enemy	30
Figure 15. Level 2 Minor Enemy	30
Figure 16 Level 3 Major Enemy	31
Figure 17. Level 3 Minor Enemy	31
Figure 18. Level 4 Major Enemy	31
Figure 19. Level 4 Minor Enemy	31
Figure 20. Level 5 Major Enemy	32
Figure 21. Level 5 Minor Enemy	32
Figure 22. Class Diagram	34
Figure 23. State chart diagram of GameScreenManager object.....	39
Figure 24. New game sequence diagram	42
Figure 25. Load game sequence diagram.....	45
Figure 26. Pause & Save game sequence diagram	48
Figure 27. Activity Diagram	49
Figure 28. Basic Subsystem Decomposition.....	60
Figure 29. Detailed Subsystem Decomposition.....	61
Figure 30. User Interface Subsystem.....	62
Figure 31. Game Management Subsystem	64
Figure 32. Game Entities Subsystem	65
Figure 33. Hiearhical layers of system.....	66

1. Introduction

“Split-Field” is a platform game which is consisted of different number of levels. The game success depends on player’s quick time responses, strategic capabilities and enemy tracking. Split-Field’s main goal is to push players’ skills to the limits while aiming them to have fun.

In Split-field there are different number of levels. In each level there is a major enemy and a few minor enemies with it. Our main character is a spaceship named Line-Rider. It can move on the outer boundaries of the field. On the map, there are also power-up boxes, which give different variety of enhancement to the Line-Rider. These power-up boxes are not mandatory, but optional. The player is able to finish the game without even using them. Using them is not affecting the score: there is no penalty. But a clever player is welcomed to use them to get higher scores in the game.

Main purpose in a level is to cut off an area while not getting touched by the major or minor enemies. To do that, Line-Rider should be directed into the field from “safe” boundary lines. After completing and reserving a portion of the field, it should return to safe boundary lines before contacting an enemy in the field. In the field, Line-Rider becomes vulnerable to enemy contact. After completing a portion-cutting operation, that field would be removed from the original field and remaining boundary becomes a “safe” boundary for our Line-Rider to move in. Player moves to next level if %80 of the field is captured.

Game is originally based upon a 90’s DOS game named “Volfied”, which is also based upon a Commodore64 game named “Qix” made by Taito Company [1]. Split-Field will be a PC game written in Java, aiming to use the object oriented model programming. It should run on every kind of operating system since Java supports all of them.

2. Requirement Analysis

2.1. Overview

As previously stated above, Split-field is a platform-action game. Game is consisted of finishing all the levels in the game without losing all of lives given at the start. Total number of levels will be determined in implementation stage of our project. The data of level numbers will be held out of program code. It will be held in a text file. Thus after its implementation, we would be able to change its design aspects without even touching to our source code.

In our game, player will be welcomed with the main menu. After selecting “Play Game” in the menu, the user is asked if they want to start a new one or continue from a previously saved game. If user selects new game, if user selects new game, level 1 will commence right away.

In every level, our main character (Line-Rider) will start from center of bottom line of the field. It will have three lives given from start. There is also a shield surrounding the Line-Rider and its capacity is designed to last in 900 seconds. This amount is illustrated in the left corner of screen and it will decrease as time flies. If remaining shield capacity reaches zero, Line-Rider will be vulnerable to all enemies whether it is on a main line or not.

While capturing an area, Line-Rider is also vulnerable to enemies. Player must make decisions between getting caught and losing a life or getting a bigger chunk of the field.

2.1.1. Control

Players are able to control our main character via using move keys on the keyboard; up arrow, down arrow, left arrow and right arrow. Additionally, users are not allowed to go into field while they are on main lines. If user decides go into field for split operation, they must

press and hold the spacebar to enter into it. After entering, holding down spacebar is not necessary to continue moving.

2.1.2. Line-Rider

Line-Rider is our spaceship, which is controlled by the Player. It has got a shield, which lasts 900 seconds before finishing the level.

Line-Rider is also capable of splitting the field in any shape player desired it to be. Its movement capability allows turning 90 degrees left or right. So any concave or convex rectangular shape could be cut from the field.



Figure 1: Playable character “Line-Rider” [2]

2.1.3. Enemies

There are different kind of enemies in every level; major enemy and minor enemies. In each level enemies vary in standards of characteristic, design and speed. After Line-Rider enters the field, they become slightly faster than their previous patrolling speed.





Figure 2 Some of major enemies from different levels [2]

2.1.4. Power-ups


There are 4 types of power-ups in every level to help player win the level. Placements of power-ups differ for every level. These power-ups are wrapped in a power-up box before they

are acquired. Splitting the area, they exist, and cutting them from original field does require them. After requiring, they are explained with a logo where they were when they were a box. Here is the list of all four power-ups:

: Clear all minor enemies from field.

: Line-Rider moves with more speed for a short time.

: Pause shield decrement for a few seconds.

: Line-Rider is able to shoot laser if spacebar is pressed. Lasers destroy only minor enemies, not major ones.

2.2. Functional Requirements

2.2.1. New Game

The player can choose “New Game” option from the play game menu to start a game from very beginning. The default parameters determined by the system will be used to construct the levels and the player will start from the first level of the default levels.

2.2.2. Play Game

In each level, the in-game character controlled by the player will be located to middle of the bottom line of the game field and, enemies and power-ups will be distributed randomly to the game field by the system. The attributes of the enemies and power-ups will vary each level. Enemies will be able to move in the game field and the control of the player character will be provided by the “arrow” keys, but power-ups will be steady throughout the game. When the player is at the edges of the game field, the enemy will not be able to harm the player, but in the game field if enemy contacts with the player the health of the player will decrease, the line player create will be destroyed

and the player will be located to the beginning point of the line. The purpose of the game is cutting the 80% of the game field before the health of the player, which can be damaged by enemies, ends. To cut an area, initially the player must press on the “space” key and then draw the lines of cutting area by “arrow” keys. The cutting line should start and end on any edge of the game field, and the other points of the cutting line should contact with the game field. Since these cutting rules create 2 areas with different sizes separated by lines, the small area will be considered as the cutting area. If there are enemies in the cutting area, the enemies will be destroyed and the player will gain score for each enemy and, if there are power-ups in the cutting area, the player will be able to use them. The player will repeat cutting the area procedure until 20% of the game field remains. If the health of the player ends before 20% of the game field is cut, the game will over and a game over message will disappear in the screen. When the 80% of the game field is cut and the health of the player is not consumed, the player will proceed to the next level which is predetermined by the system. Until the default levels are over, the system will upload next level with different backgrounds, enemies and power-ups. At the beginning of each level, the health of the player will be recharged. The same cutting and playing rules will be applied at each level but the velocity, damage and health magnitude of enemies and, the properties of power-ups will vary as well as the numbers of these game objects. If the player level up all levels by in each level cutting 80% of the game field without consuming its health, he/she will win the game and if he/she score up in high score list, the system will prompt the player to enter his/her name to update high score list.

2.2.3. Load Game

The player will be able to continue from the last saved level in addition to starting to a new game. Within the play game menu, the player can load its progress by clicking on the “Load Game” button, if there is a saved level before and resume the level.

2.2.4. Pause Game

There will be a pause icon in the corner of the game screen to pause the game. The player can pause the game any time during gameplay by clicking on that icon. The player can continue to the game, save its progress, return to the main menu or exit the game from the pause menu.

2.2.5. Save Game

When the player wants to save its progress, the system will allow user to save the last played level. Within the pause menu, the player can save its progress by clicking on the “Save Game” button in order to progress from the level he/she left off as it was detailed in above “Load Game” section.

2.2.6. Exit Game

From the main menu or the pause menu, the player can exit the game by clicking on the “Exit Game” button. Since the game will not provide automatic saving, if the player will not save his/her progress from pause menu manually, he/she will not be able to load the last level he/she played.

2.2.7. Show High Scores

The highest 10 score and their holders will be stored in descending order by the system. The player will be able to see high score table by clicking on the “Show High

Scores” button from main menu. If any player has never played the game before, the system will show “No High Score” message to player.

2.2.8. Show Help

The player can obtain information about the game from the main menu. This information will contain player control, the features of the power-ups and other main rules about the game to learn how to play the game.

2.3. Non-Functional Requirements

- **Performance**
 - No other game engine will be used to design the game other than Java GUI itself, so not using complex graphics will reduce the response time of the system.
 - As the Split-Field save data to .txt files or load data from .txt files locally, the response time of storage operations will be more time efficient comparing to server-based data storage.
- **Usability**
 - The game must have a consistent and user- friendly interface which will provide familiar control environment to the user.
 - The player can exit the game from any menu he/she encountered through the game.
- **Clarity and Smoothness**
 - To provide a satisfactory user experience, the game must have a clear and smooth graphics which will be provided by setting minimum frame rate as twenty-five frame per second.

2.4. Constraints

- The project will be written using Java programming language.
- GitHub, web-based hosting system, will be used to host project codes.
- The group member will use Eclipse development environment for the implementation of the project.
- Graphic objects such as player character and enemies will be designed using Adobe® Photoshop CS6.

2.5. Scenarios

Use cases will be derived from the following scenarios:

1. **Scenario Name:** playsAndFinishesGame

Participating actor instances: bob:Player

Entry condition: Bob opens the game.

Exit conditions: Bob exits.

Flow of events:

- 1) Bob selects “Play game”.
- 2) Bob chooses one of two options which are starting from first level and continuing from last saved level.
- 3) Bob cuts game field in each level until he cuts the %80 of the area and win the game or the enemy touches three times to player during the game and enemy defeats Bob.
- 4) Game finishes and Bob is asked to write a name if his score is in top ten high scores.

2. Scenario Name: saveTheGame

Participating actor instances: bob:Player

Entry condition: Bob opens the game.

Exit conditions: Bob exits.

Flow of events:

- 1) Bob selects “Play game”.
- 2) Bob chooses one of two options which are starting from first level and continuing from last saved level.
- 3) Bob pauses the game and sees an in-game menu.
- 4) Bob selects “Save”.

3. Scenario Name: viewHelp

Participating actor instances: bob:Player

Entry condition: Bob opens the game.

Exit conditions: Bob exits.

Flow of events:

- 1) Bob selects “Help”.

4. Scenario Name: viewHighScores

Participating actor instances: bob:Player

Entry condition: Bob opens the game.

Exit conditions: Bob exits.

Flow of events:

- 1) Bob selects “High Scores”.

2.6. Use Case Models

The possible user interactions and use cases of the player with the system is briefly displayed in figure 1 as a use case diagram. Individual use cases will also be described textually after the diagram.

Visual Paradigm Standard Edition (Bilkent Univ.)

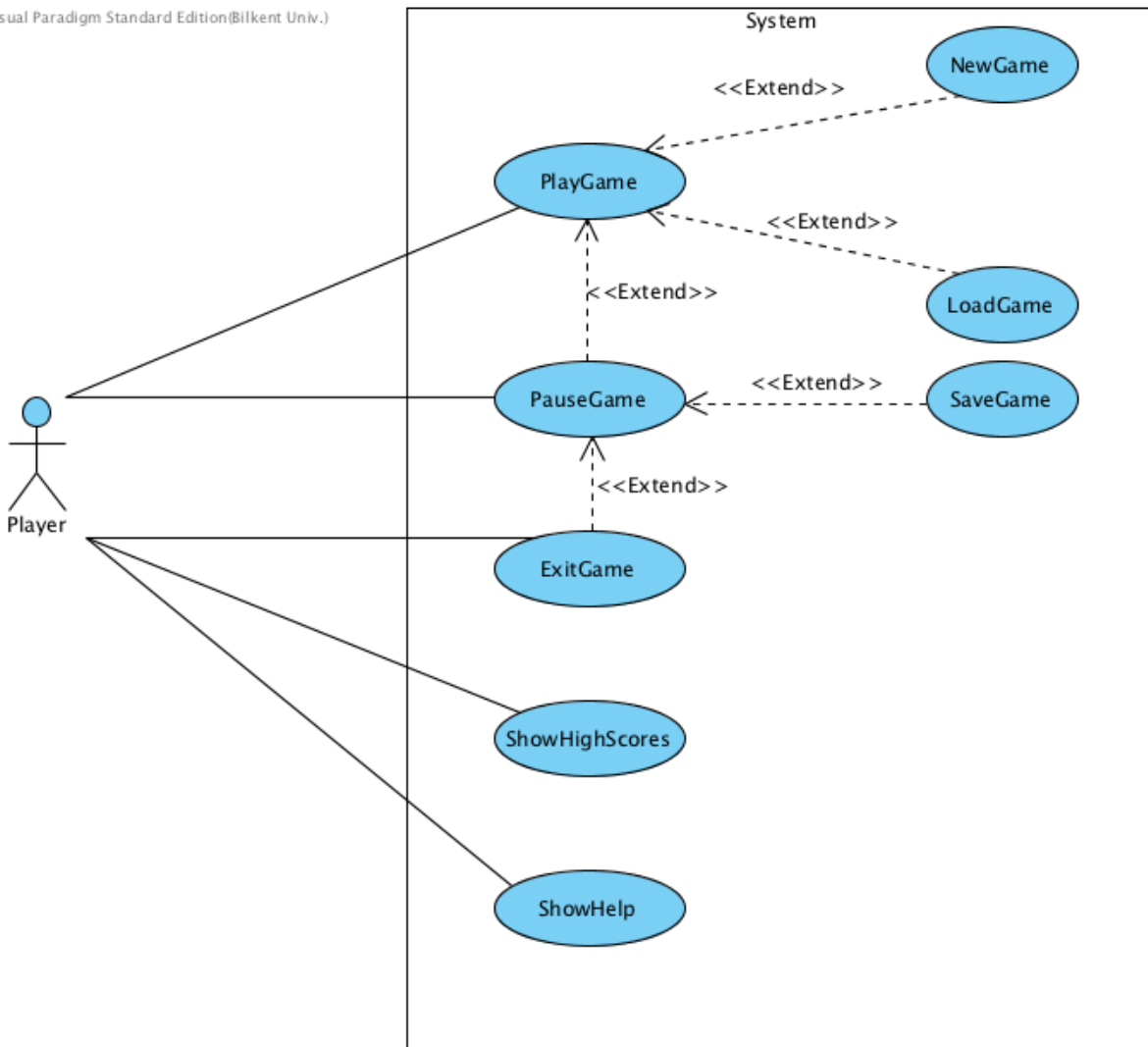


Figure 3. Use Case Diagram

2.6.1. Use Case Descriptions

Use case #1

Use case name: Play game

Participating actors: Player

Pre-condition: The player has already entered main menu.

Post-condition: The system directs player to a new screen that contains new game and load game buttons.

Entry condition: The player clicks the “Play Game” button.

Flow of events:

1. The system displays a menu screen that contains new game and load game buttons.

Exit condition:

- The player chooses to return to main menu.

Exceptions:

- The player returns to main menu.
 - The player clicks on the return button from menu.

Use case #2

Use case name: New game

Participating actors: Player

Pre-condition: Player has already entered the play game menu.

Post-condition: The system shows the high score table.

Entry condition: The player clicks on the new game button.

Flow of events:

1. The system constructs the levels which are determined in “Levels.txt” file by the system.
2. The player encounters with the level.
3. The player starts from middle of the bottom line of the game field.
4. The player cuts the game field until 80% of it is cut.
5. The system directs player into the next level.
 - a. Steps 3, 4, 5, 6 are repeated until the game is over.
6. The system displays the score of the player. If player’s score is in the top ten scores, the system asks player to write his/her name.
7. The player writes his/her name.
8. The system saves the score and the name, and does modification on the high score table.
9. The system shows the high score table.

Exit condition:

- The player has finished all the levels successfully.
- The player has lost its lives throughout the game before finishing the all the levels successfully.
- The player has clicked pause button and choose to return to the main menu.

Exceptions:

- The player loses all its lives throughout the game before finishing the all the levels successfully.
 1. The player starts with the level.
 2. During the area cutting operation, an enemy touches the player.
 3. The system decrements the number of player’s lives.
 - a. Steps 1, 2 and 3 are repeated until player loses all of his/her lives.
 4. The system shows a game over message to player.
- The player can pause the game at any time and return to the main menu.

Quality Requirements: At any point during the flow of events, this use case can include the PauseGame use case.

Use case #3

Use case name: Load game

Participating actors: Player

Pre-condition: The player has already entered play menu and there is already a saved state.

Post-condition: The player continues playing the last saved level.

Entry condition: The player clicks on the load game button.

Flow of events:

1. The system constructs the last saved level and rest of the higher levels from “LastSavedLevel.txt” file.
2. The player continues playing the last saved level.

Exit condition:

- The player starts to play the game from the last saved level.

Exceptions:

- If there is no saved state, the player encounters with the first level of the predetermined level set by the system.

Use case #4

Use case name: Pause Game

Participating actors: Player

Pre-condition: Player has already been playing the game.

Post-condition: -

Entry condition: The player clicks on the pause button during the gameplay.

Flow of events:

1. The system displays the in-game menu.
2. The player chooses one of the options in-game menu by clicking buttons which are “Continue”, “Save Game”, “Return to Main Menu” and “Exit”.

Exit condition:

- The player returns to the main menu by clicking on the “Return to Main Menu” button.
- The player returns to the gameplay by clicking on the “Continue” button.

Exceptions: -

Use case #5

Use case name: Save Game

Participating actors: Player

Pre-condition:

1. The player has already been playing the game.
2. The player has clicked on the “Pause” button to be directed in-game menu by the system.

Post-condition: The system displays “Saved Successfully!” to the player.

Entry condition: The player clicks on the “Save” button from in-game menu.

Flow of events:

1. The system modifies the “LastSavedLevel.txt” file which contains the number of the last saved level.
2. The system displays the message that the game is saved successfully without interrupting the game.

Exit condition: The player clicks on the “Continue” or “Return to Main Menu” buttons from the in-game menu after the game is successfully saved.

Exceptions: None

Use case #6

Use case name: Exit Game

Participating actors: Player

Pre-condition: The player has already entered the main menu or in-game menu by clicking on the “Pause” button during gameplay.

Post-condition: The player exits the game.

Entry condition: The player clicks on the “Exit” button.

Flow of events:

1. The system prompts “Are you sure?” and waits for a click from the player on “Yes” or “No” buttons.
2. Player clicks on “Yes”.
3. The system closes the main frame and terminates itself.

Exit condition: None.

Exceptions: The player clicks on the “No” button and exists the game.

Use case #7

Use case name: View High Scores

Participating actors: Player

Pre-condition: Player has already entered main menu.

Post-condition: -

Entry condition: The player clicks on the “High Scores” button that exists in the main menu.

Flow of events:

1. The system reads the “HighScores.txt” file which contains the high scores and their holders.
2. The system displays the high scores in descending order with their holders.

Exit condition:

- The player returns to the main menu by clicking on the “Return” button.

Exceptions:

- If any player has never played the game before, the system cannot display any high score and its holder. Thus, the system shows “No High Score” message to player.

Use case #8

Use case name: View Help

Participating actors: Player

Pre-condition: The player has already entered main menu.

Post-condition: The system shows a screen that has information about the game and instructions.

Entry condition: The player clicks on the “Help” button.

Flow of events:

1. The system displays information about the game and instructions.

Exit condition:

- The player returns to the main menu by clicking on the “Return” button.

Exceptions: None.

2.7. User Interface

2.7.1. Navigational Path

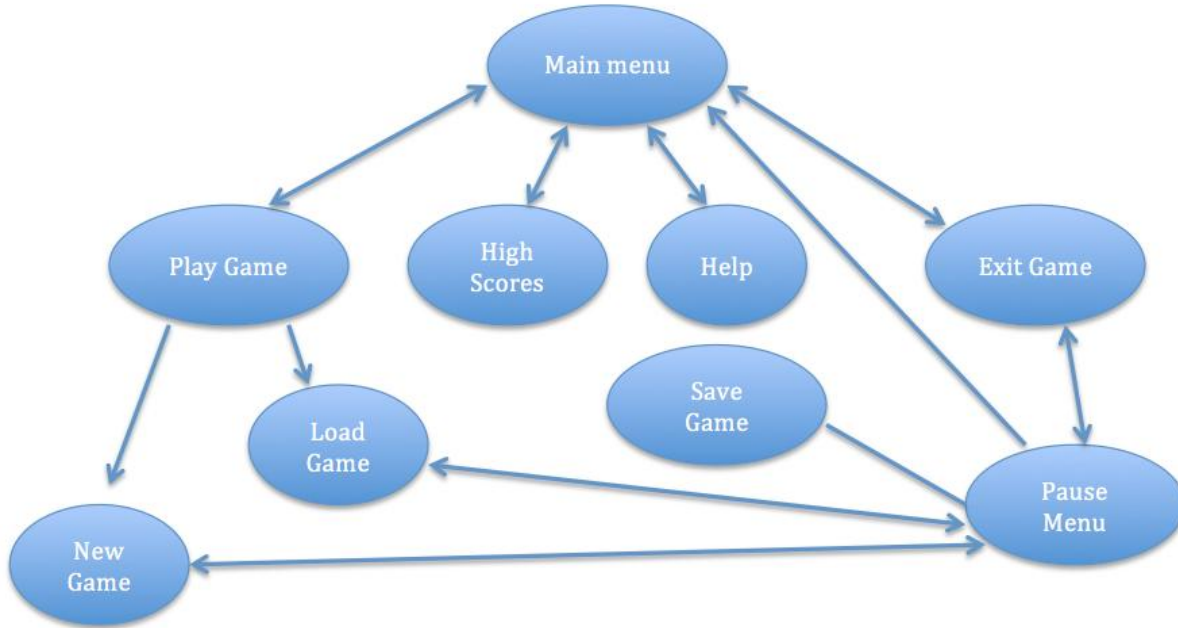


Figure 4. The navigational paths between the game menus

Since when the player chooses the save game option, the system will display a pop-up message and the user will have no control over navigation between pause menu and save game, the non-directional edge is used to indicate the relation between nodes. When the user has control over navigation, two-directional edges are used to indicate the relations between nodes. If the user chooses the exit game option, the system will prompt a pop-up message "Are you sure?" with "yes" or "no" options. If the user chooses yes option, he will exit from the game, otherwise he will return to current menu.

2.7.2. Game Menu Mock-ups

The menu designs in this section are not final and their design may change at the final version of the project.

2.7.2.1. *Main Menu*



Figure 5. The mock-up of the main menu

2.7.2.2. *High Scores Menu*



Figure 6. The mock-up of the high scores menu.

2.7.2.3. *Help Menu*

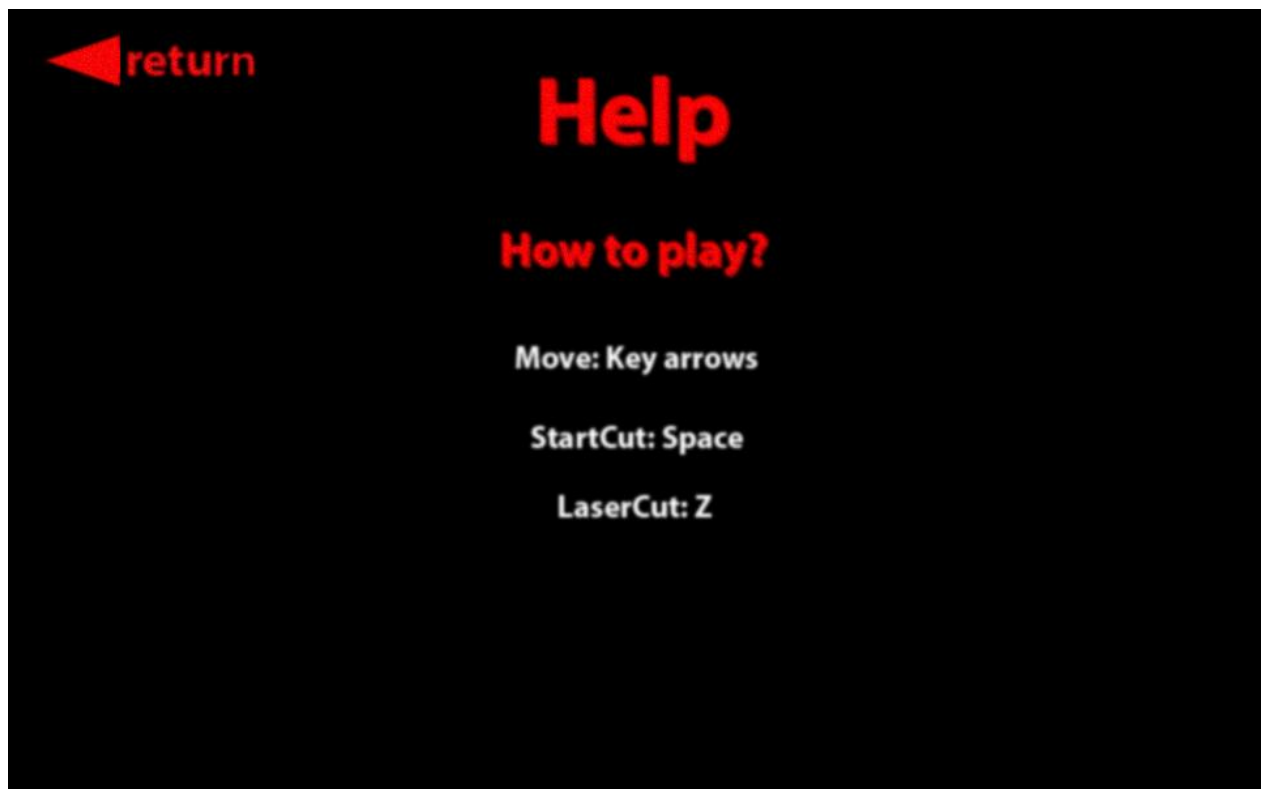


Figure 7 The mock-up of the help menu

2.7.2.4. *Play Game Menu*

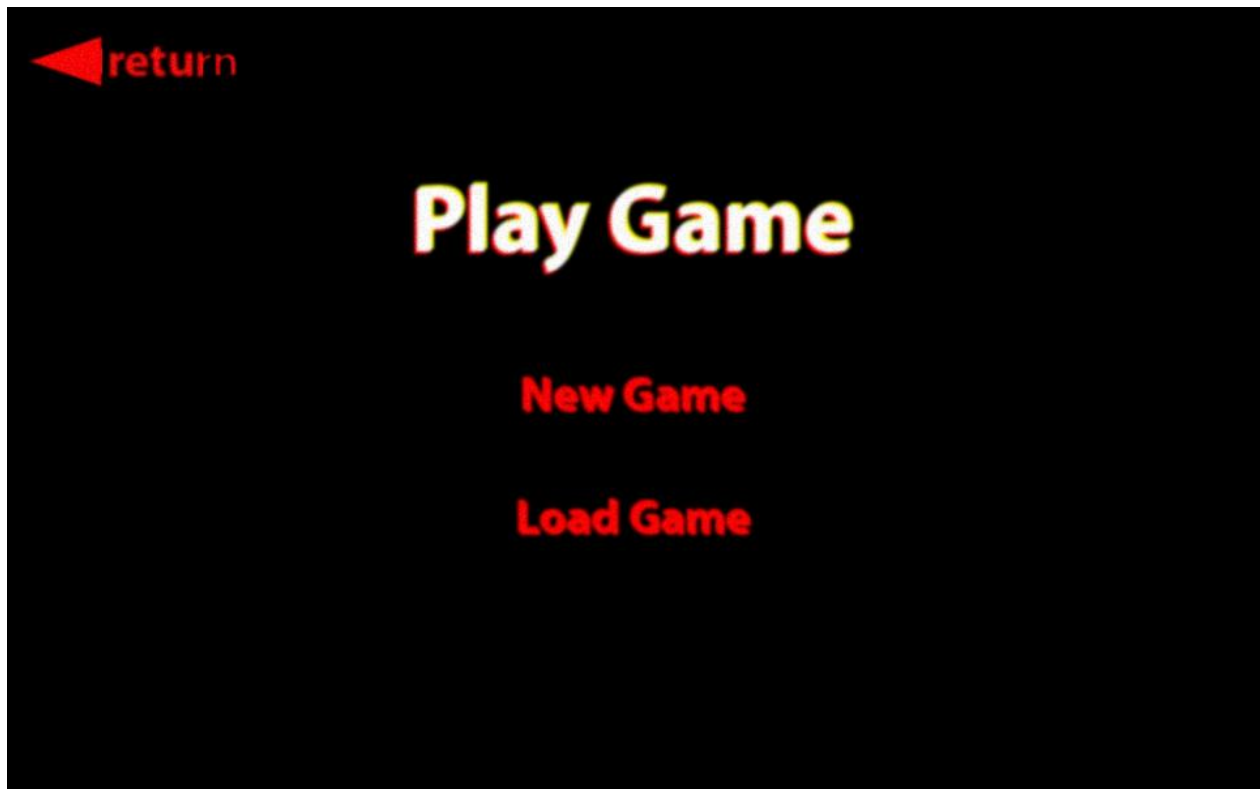


Figure 8 The mock-up of the play game menu

2.7.2.5. *In-game Menu*



Figure 9 The mock-up of the play game menu

2.7.3. In Game Graphics

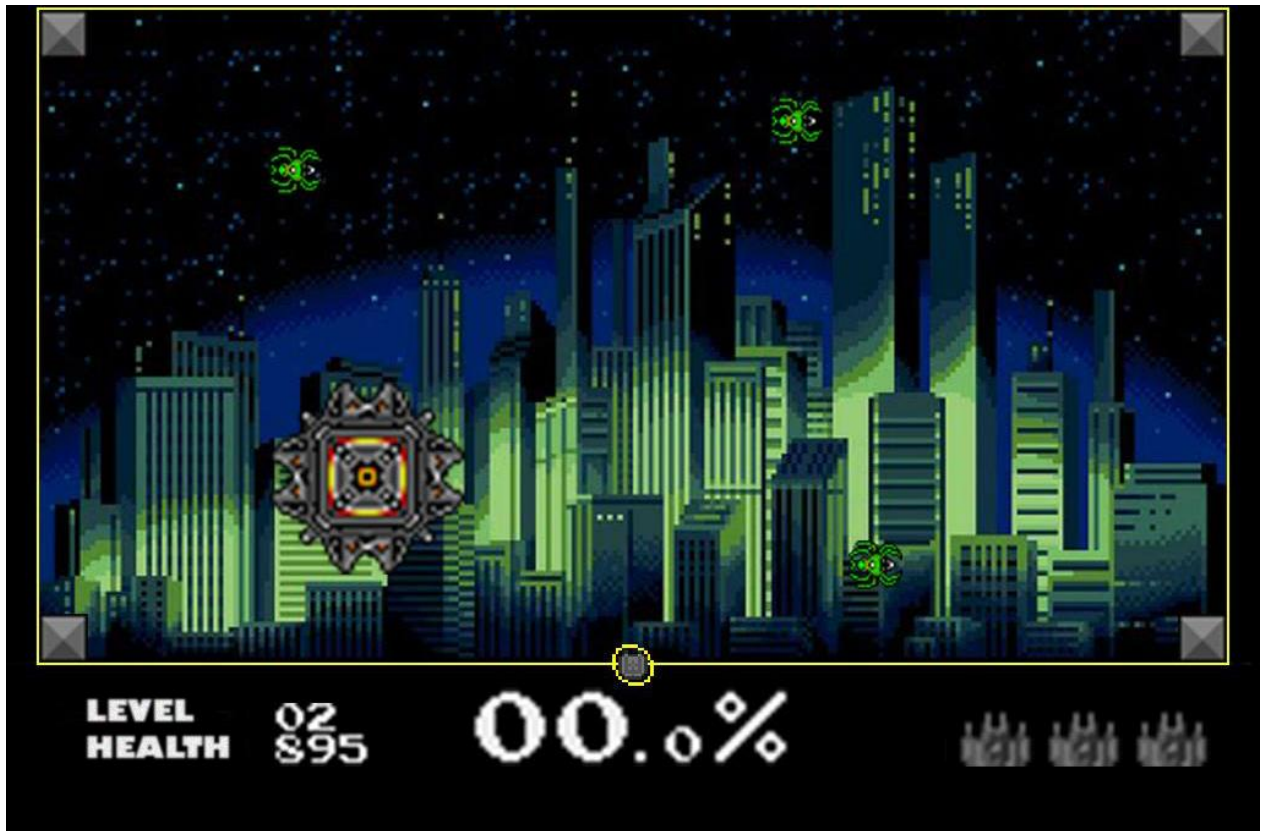


Figure 10. The mock-up of the game field

2.7.4. Characters

2.7.4.1. Line-Rider

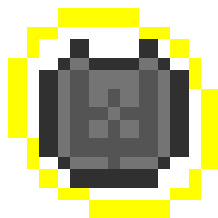


Figure 11. LineRider

2.7.4.2. Enemies

- Level 1 Major Enemy

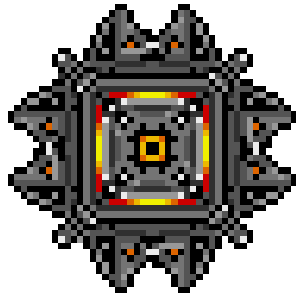


Figure 12. Level 1 Major Enemy

- Level 1 Minor Enemy

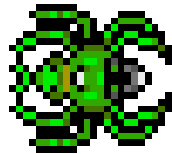


Figure 13 Level 1 Minor Enemy

- Level 2 Major Enemy

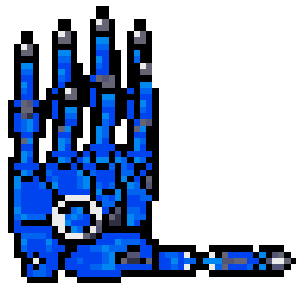


Figure 14. Level 2 Major Enemy

- Level 2 Minor Enemy

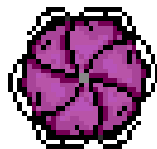


Figure 15. Level 2 Minor Enemy

- Level 3 Major Enemy

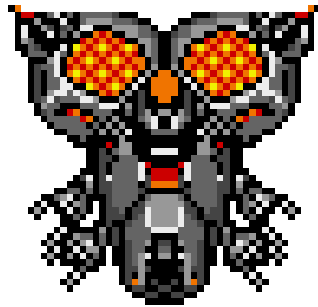


Figure 16 Level 3 Major Enemy

- Level 3 Minor Enemy



Figure 17. Level 3 Minor Enemy

- Level 4 Major Enemy

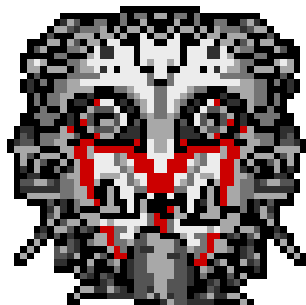


Figure 18. Level 4 Major Enemy

- Level 4 Minor Enemy

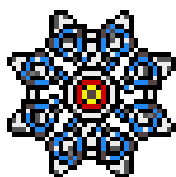


Figure 19. Level 4 Minor Enemy

- Level 5 Major Enemy



Figure 20. Level 5 Major Enemy

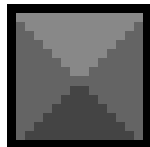
- Level 5 Minor Enemy







Figure 21. Level 5 Minor Enemy

2.7.4.3. Power-Ups

- Power-up box



- : Clear all minor enemies from field.
- : Line-Rider moves with more speed for a short time.
- : Pause shield decrement for a few seconds.
- : Line-Rider is able to shoot laser if spacebar is pressed. Lasers destroy only minor enemies, not major ones.

3. Analysis

3.1. Object Model

3.1.1. Domain Lexicon

- **Split-Field:** An arcade game. It is the name of our game.
- **LineRider:** A player character whose actions are controlled by the player.
- **Health:** An attribute assigned to entities that indicates its state in combat.
- **Enemy:** A computer character whose actions are controlled by the system algorithm.
- **Power-ups:** Objects that add extra abilities to the game character.
- **GUI:** Guided User Interface. It is the interface that enables interaction between the user and the system.
- **MVC:** Model/View/Controller. It is an architectural style that is used for designing systems and subsystems.
- **JVM:** Java Virtual Machine which is an abstract computing machine that enables a computer to run a Java program.

3.1.2. Class Diagram

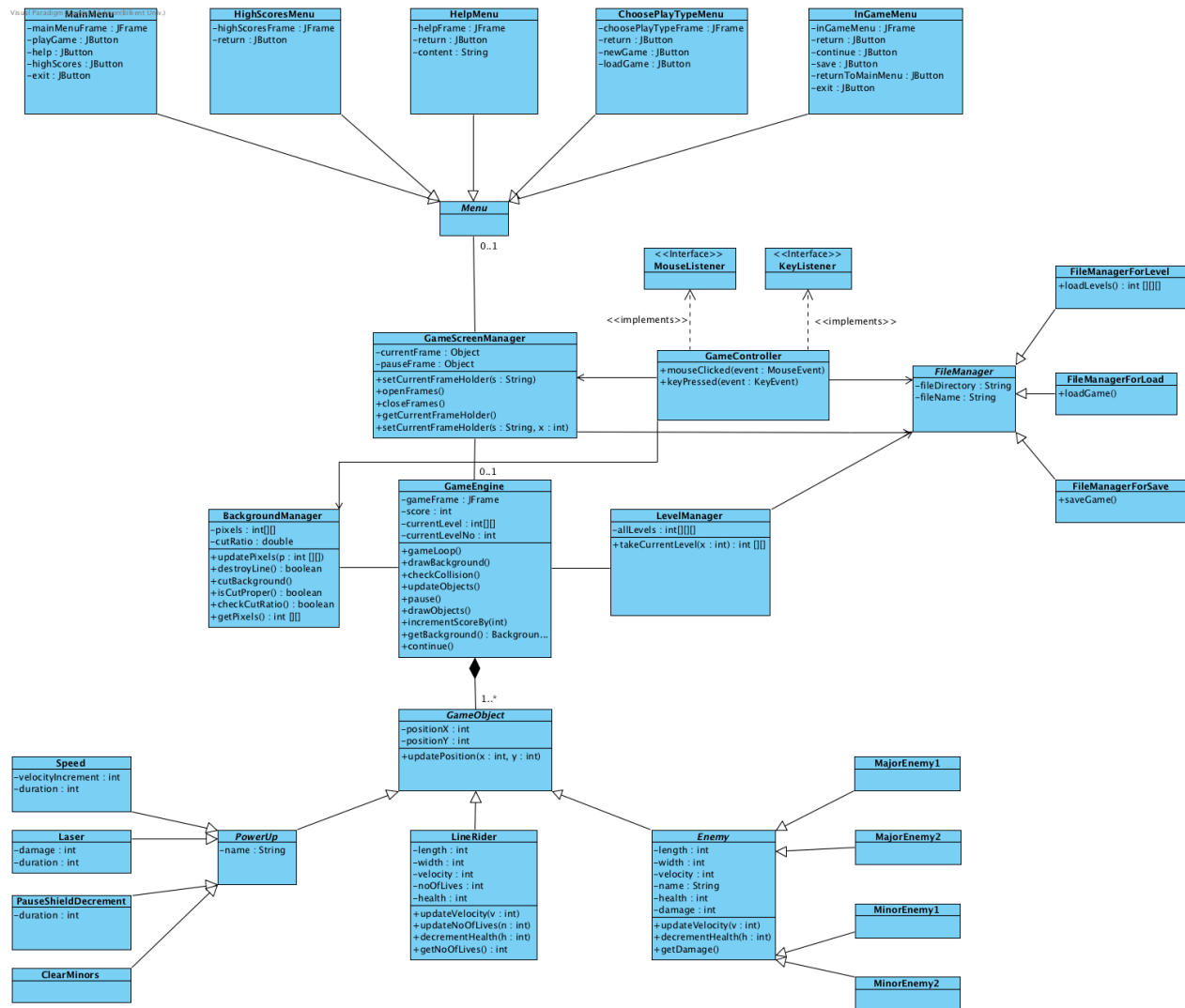


Figure 22. Class Diagram

Class diagram of “Split-Field” is illustrated above the system comprises of 26 classes whose functionalities are as follows:

Menu is an abstract class which consists of general menu attributes and operations. The following classes extend this class for carrying on necessary menu operations and forming GUI of the menus.

- **MainMenu**
- **HighScoresMenu**
- **HelpMenu**
- **ChoosePlayTypeMenu**
- **InGameMenu**

GameScreenManager class is similar to the view of MVC design pattern but it does not draw models as it is done in MVC. GameScreenManager manages the objects that draw the required frames like menus and gameplay (that is done in GameEngine class). It opens, closes and switches the previously stated objects.

Controller class is similar to the controller of MVC design pattern. It holds one GameScreenManager instance and controls the menu or gameplay object, which are drawers of frames and are in GameScreenManager instance, with inputs from the player. For example, player opens the game and sees the main menu. When it clicks a button, “mouseEvent” function is called and required operation is done on the GameScreenManager instance (e.g. changing the current frame with “setCurrentFrameHolder” function of GameScreenManager class). Or, when GameScreenManager object holds a GameEngine object, player gives the input from the

keyboard. “keyPressed” function of Controller class is called and this updates the two-dimensional integer array in the Background class. In other words, controller holds an GameScreenManager instance as the viewer and holds an Background instance to change the state of background of the game. Updated background is redrawn by the GameEngine object of the GameScreenManager class. In addition, Controller class will save the last level played by the player if player pauses the game and clicks on the save button.

FileManager is an abstract class which consists of general file manager attributes and operations. The following classes extend this class for carrying on necessary file management operations.

- **FileManagerForLevel:** This class is used by GameEngine class to load existing levels from “Levels.txt” file which holds many levels with their content (e.g. types of enemies, number of enemy for each type).
- **FileManagerForSaveLoad:** This class is used by the Controller class when player pauses the game and clicks on the save button in in-game menu. “saveGame” function saves the last level played by the player in “LastSavedLevel.txt” file.

The other function of this class, “loadGame” is called when player clicks on the load game button in menu where the player chooses playing new game or continuing game from last saver’s level.

Background class is used by GameEngine class to locate player in gameplay as LineRider object and other game objects. Gamefield is represented as two-dimensional integer array in this class. Different integers represent the different game objects, the split area and borders. Two-dimensional integer array is updated by Controller class when player directs LineRider from keyboard. Updating two-dimensional integer array is done with “updatePixels” function.

GameEngine uses “isCutProper” function to control the line drawn by LineRider is proper for cutting. Then it calls “cutBackground” to split cut area from background. If an enemy touches the line drawn by LineRider when player has not finished drawing, GameEngine calls the “destroyLine” function. Finally, being successful at level is checked by GameEngine via “checkCutRatio” function.

GameEngine class is the most important class of the game in which all the game objects are declared according to the information of levels taken from the “Levels.txt” with the function “takeCurrentLevel” of LevelManager class. Graphics are drawn and redrawn with “drawObjects” and “drawBackground” functions according to the information taken from game objects and background. States of game objects are updated in this class with “updateObjects” function in which update functions of all game objects are called. It also, conducts the game in a loop called “gameLoop” that is basically makes the game go on as long as end conditions, which are cutting backgrounds of all levels successfully or killing certain time during the game, are not reached.

LevelManager class is used by GameEngine to obtain the level informations according to the current level. For example, when player pass the Level 2, GameEngine calls the “takeCurrentLevel” function of LevelManager which returns a two-dimensional integer array.

GameObject abstract class represents all of the items seen by the player. It holds the coordinates of the items as the x-position and y-position. Update of this coordinates are done in the “gameLoop” function of the GameEngine class.

- **LineRider** class
- **PowerUp** abstract class

- **Enemy** abstract class

inherit and expand the GameObject class. Required functions and attributes are added to these classes.

- **Speed**
- **Laser**
- **PauseShieldDecrement**
- **ClearMinors**

classes inherit the PowerUp abstract class. They represent the power ups of the LineRider in the gameplay.

- **MinorEnemy1**
- **MinorEnemy2**
- **MajorEnemy1**
- **MajorEnemy2**

classes inherit the Enemy abstract class. They represent the enemies in the gameplay. They are different from each other with their appearance, velocity, health and damage.

3.2. Dynamic Models

3.2.1. State Chart

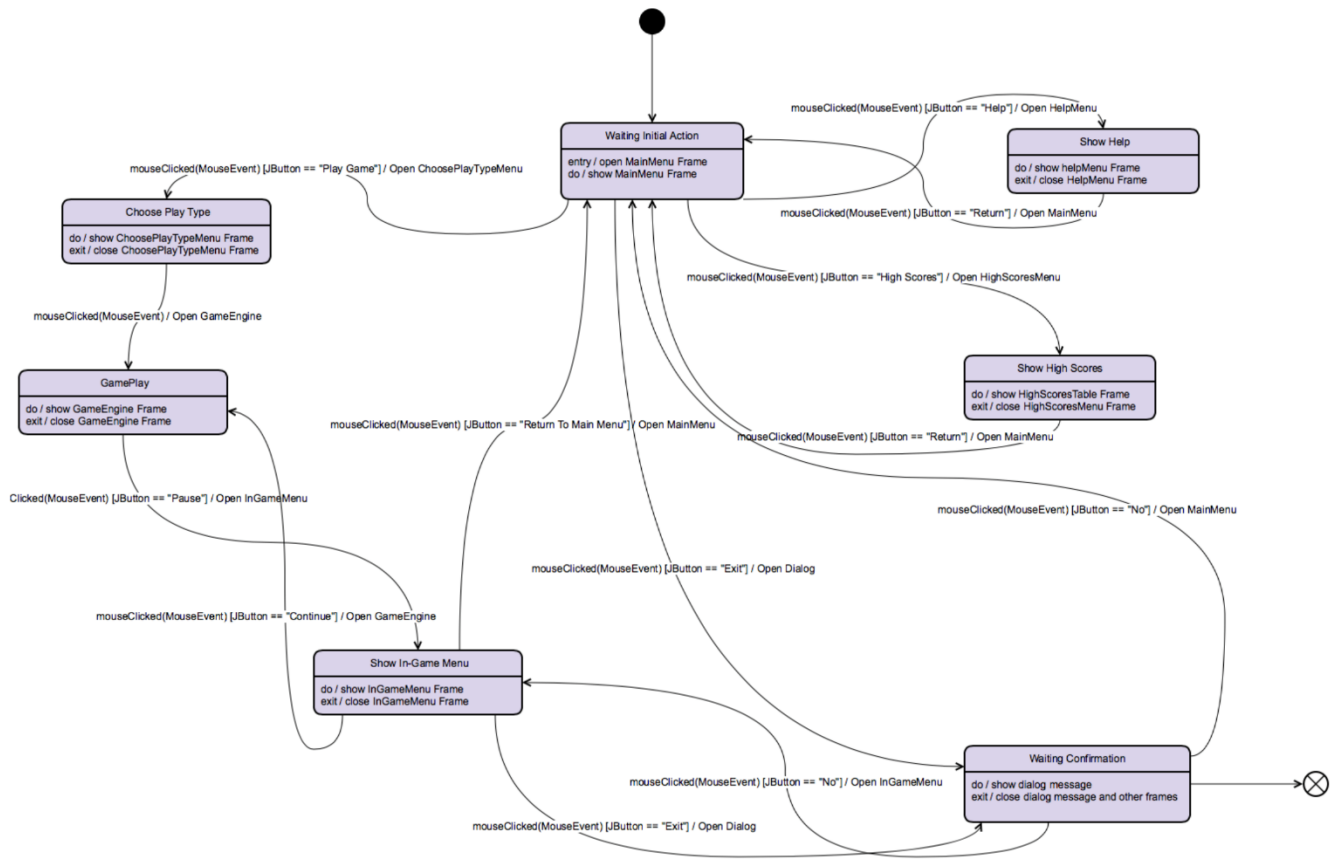


Figure 23. State chart diagram of GameScreenManager object.

3.2.2. Sequence Diagram

After the player runs the game, the main menu will appear on the screen and when the player clicked on the “Play Game” button, two options which are “New Game” and “Load Game” will appear on a new frame. If the player chooses to play new game the below scenario will be applied.

3.2.2.1. New Game

Scenario: The player chooses the “New Game” from play menu. After that, the mouseClicked() function of Controller object will be called and according to the button clicked on by the player, setCurrentFrameHolder() function of GameScreenManager object of Controller will be called with a parameter which indicates the choice of player which will be “New Game” in this context. setCurrentFrameHolder() will first call the closeFrames() function of GameScreenManager class then initialize a new GameEngine object and assign it as its currentFrame. The constructor of GameEngine will first assign the integer parameter to “currentLevelNo” variable which indicates the level, which will be 1 in this context because player chose “New Game”, player wants to start, then use takeCurrentLevel() function of LevelManager object of itself. LevelManager will have already had all levels stored as three-dimensional integer array. takeCurrentLevel will return the allLevels[currentLevel][][] array. Then GameEngine constructor will declare objects with level information taken from the two-dimensional array. A Background object will be declared in the GameEngine constructor. Created GameObject objects will be located on the background with the updatePixels() function of Background object of GameEngine. Controller calls the getBackground() function of GameEngine object which will be taken with the getCurrentFrameHolder() function of GameScreenManager object of Controller. Gameplay will be ready for player. Controller

takes the keyboard input from the user with `keyPressed()` function and updates the location of player(or `LineRider`) in the two-dimensional integer array with the `updatePixels()` of `Background` object of `Controller`. `gameLoop()` function will go on until the end conditions will be reached. This function will call the `drawObjects()` and `drawBackground()` functions of `GameEngine` with the help of two-dimensional integer array taken from the `Background` object. `GameEngine` will control the appropriateness of cuts with `isCutProper()` function of `Background` object and if it is proper, `GameEngine` will call the `cutBackground()` function of `Background`. Also, in the same function, collision of enemy and player will be checked with `checkCollision()` function of itself. If a collision occurs, `GameEngine` will decrement player's health with `decrementHealth()` function of `LineRider`. In each turn of game loop, `checkCutRatio()` of the `Background` will be called to continue to loop(or game). This scenario's sequence diagram can be seen in Figure 5.

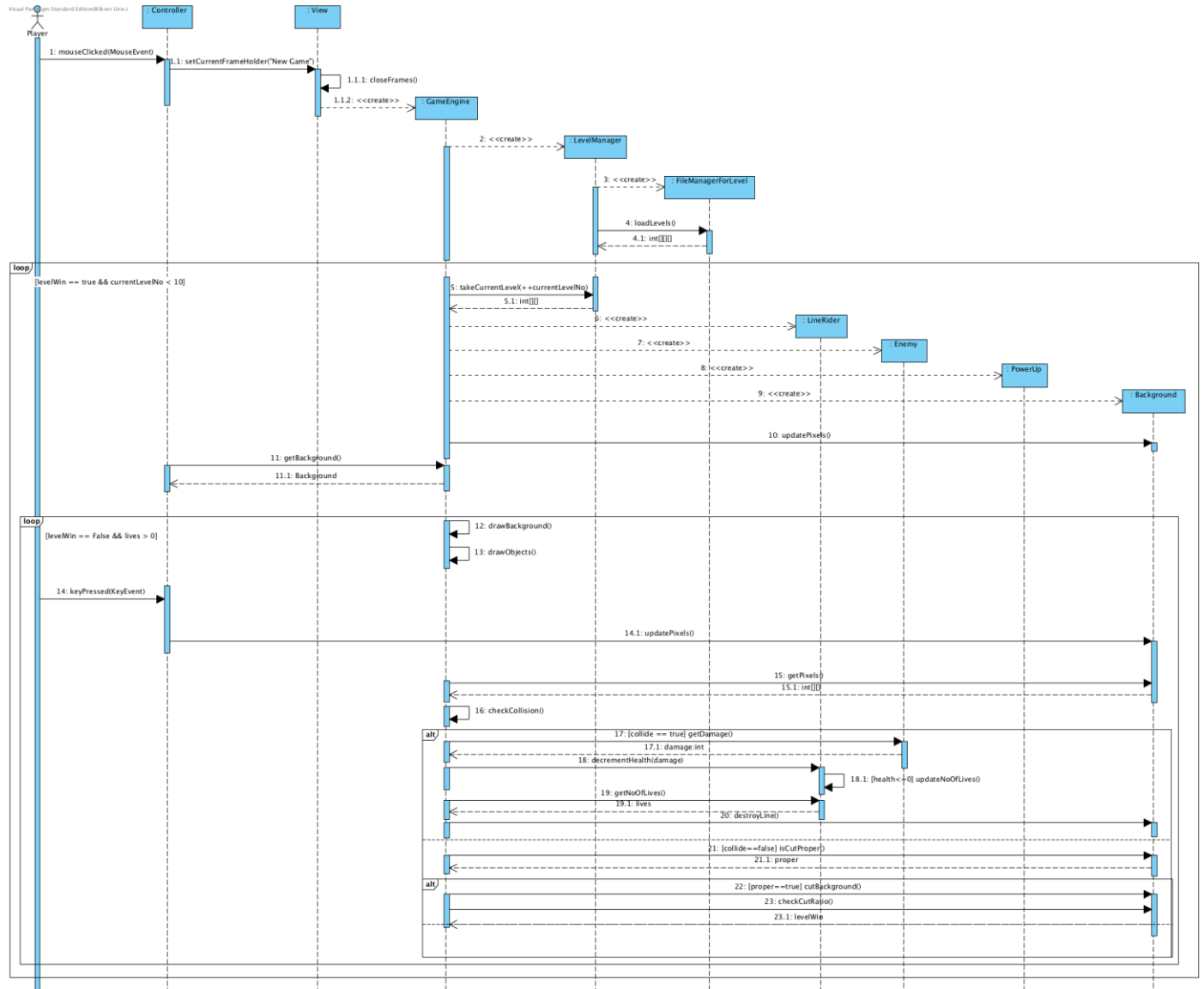


Figure 24. New game sequence diagram

3.2.2.2. Load Game

Scenario: The player chooses the “Load Game” from play menu. After that, the mouseClicked() function of Controller object will be called and because clicked on button will be “Load Game”, Controller will call the loadGame() function of FileManagerForSaveLoad object to get the last saved level. setCurrentFrameHolder() function of GameScreenManager object of Controller will be called with two parameters which indicate the choice of player, which will be “Load Game” and the level desired to start in this context. setCurrentFrameHolder() will first call the closeFrames function of GameScreenManager class then initialize a new GameEngine object and assign it as its currentFrame. The constructor of GameEngine will first assign the integer parameter to “currentLevelNo” variable which indicates the level player wants to start, then use takeCurrentLevel() function with an integer parameter which will indicate the desired number of level to take from LevelManager object. LevelManager will have already had all levels stored as three-dimensional integer array. takeCurrentLevel() function will return the two-dimensional integer array. Then GameEngine constructor will declare objects with level information taken from the two-dimensional array. A Background object will be declared in the GameEngine constructor. Created GameObject objects will be located on the background with the updatePixels() function of Background object of GameEngine. Controller calls the getBackground() function of GameEngine object which will be taken with the getCurrentFrameHolder() function of GameScreenManager object of Controller. Gameplay will be ready for player. Controller takes the keyboard input from the user with keyPressed() function and updates the location of player(or LineRider) in the two-dimensional integer array with the updatePixels() of Background object of Controller. gameLoop() function will go on until the end conditions will be reached. This function will call the drawObjects() and

drawBackground() functions of GameEngine with the help of two-dimensional integer array taken from the Background object. GameEngine will control the appropriateness of cuts with isCutProper() function of Background object and if it is proper, GameEngine will call the cutBackground() function of Background. Also, in the same function, collision of enemy and player will be checked with checkCollision() function of itself. If a collision occurs, GameEngine will decrement player's health with decrementHealth() function of LineRider. In each turn of game loop, checkCutRatio() of the Background will be called to continue to loop(or game). This scenario's sequence diagram can be seen in Figure 5.

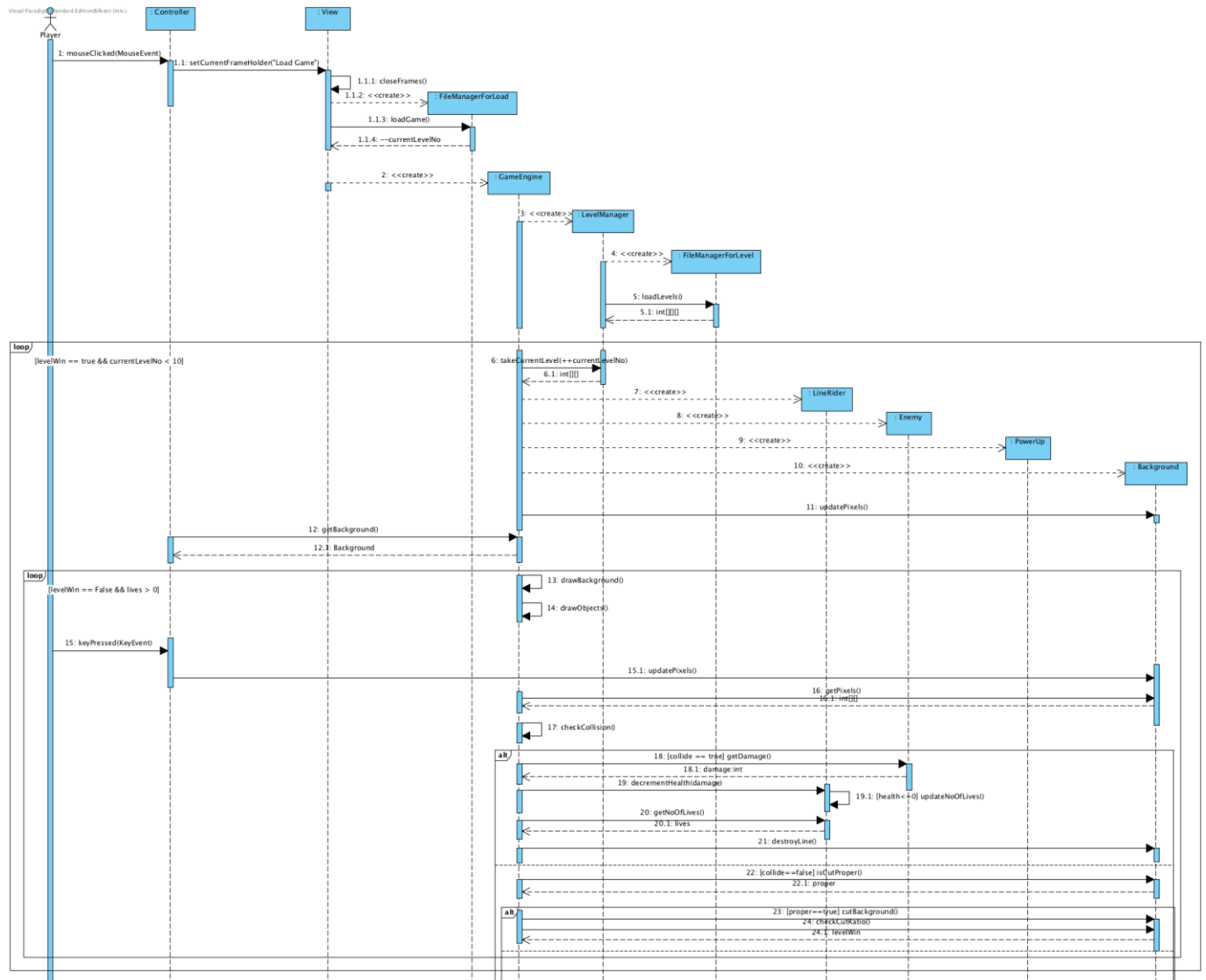


Figure 25. Load game sequence diagram

3.2.2.3. Pause & Save Game

Scenario: As long as game loop continues to be executed and the player can control the flow of the game, the player can click on the pause button that will be located top right corner of the screen to interrupt flow of the game. When the button is clicked, the mouseClicked() function of Controller object will be called and because clicked on button will be “In-game menu”. The setCurrentFrameHolder() function of the GameScreenManager object will set the current frame as inGameMenu frame. Then the gameEngine frame will be closed by closeFrames() function of the GameScreenManager object. Then the GameScreenManager object will call pause() function and the game will pause. The GameScreenManager object will initiate InGameMenu object and in-game menu which consists of 4 options, which are “Save”, “Load”, “Return to Main Menu” and “Exit”, will appear on the screen. If the player wants to save his game, he will click on the “Save” button and the mouseClicked() function of Controller object will be called. Then the Controller object will call saveGame() function of the FileManagerForSave class. The saveGame() function will modify LastSavedLevel.txt file by writing the current level that the user is playing. When the game has been saved this function will display a “Successfully Saved” message on the screen and the player will continue to see in-game menu. If he clicks on the “Continue” button, the mouseClicked() function of Controller object will be called and the inGameMenu frame will be destroyed by closeFrames() function of GameScreenManager object, so in-game menu will disappear. Then by setCurrentFrameHolder() function of the GameScreenManager object, the current frame will be gameFrame and the player will continue to the game from where he left. If he clicks on the “Return to Main Menu” button, the mouseClicked() function of Controller object will be called and the current frame will be closed by closeFrame() function of the GameScreenManager object. Then by setCurrentFrameHolder() function of the

GameScreenManager object, the current frame will be mainMenuFrame. Then the GameScreenManager object will initiate MainMenu object. If he clicks on the “Exit” button, the mouseClicked() function of Controller object will be called and Controller object will call java.lang.System.exit() method that terminates the currently running JVM. This scenario’s sequence diagram can be seen in Figure 6.

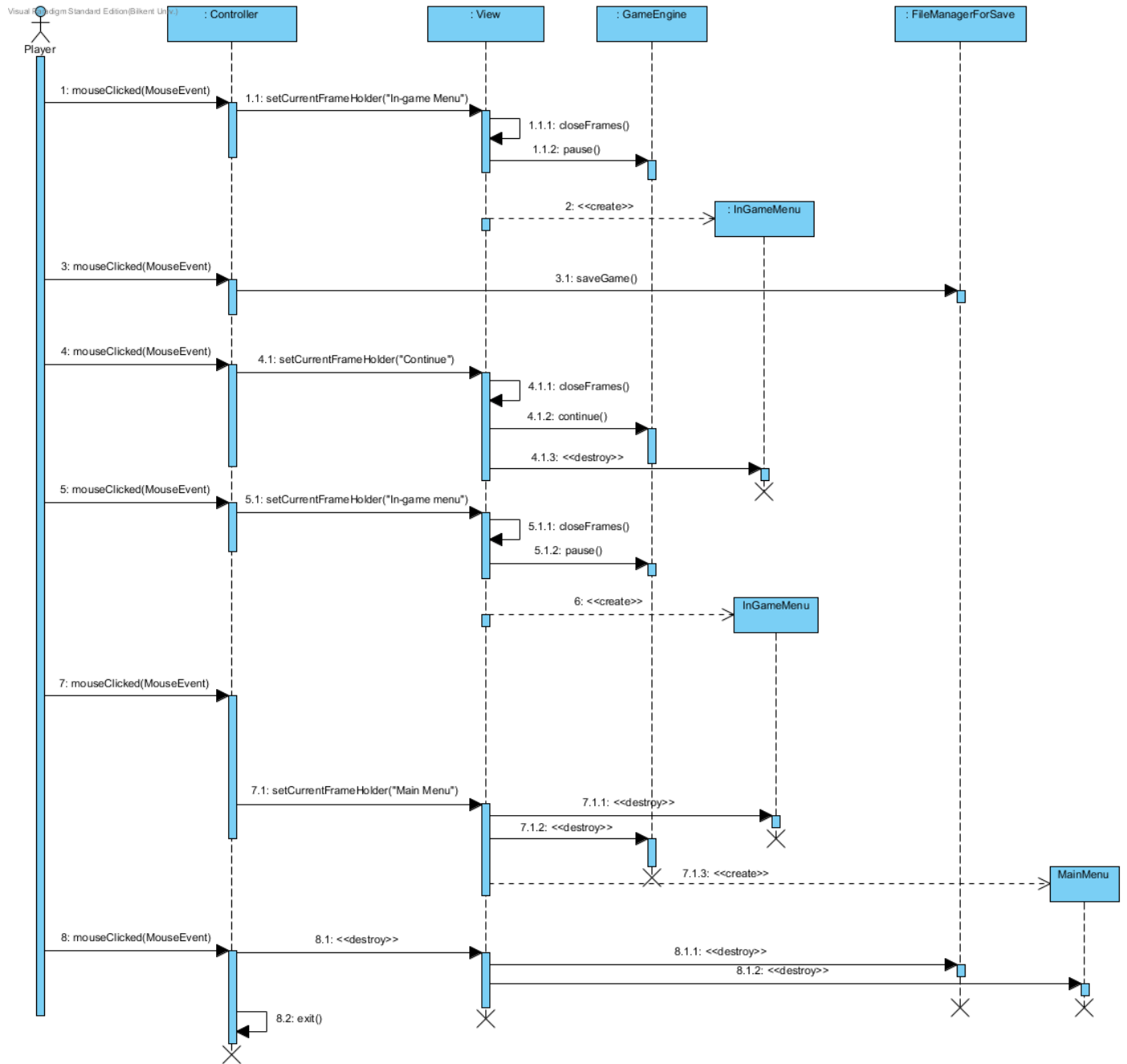


Figure 26. Pause & Save game sequence diagram

3.2.3. Activity Diagram

Visual Paradigm Standard Edition (Bilrent Univ.)

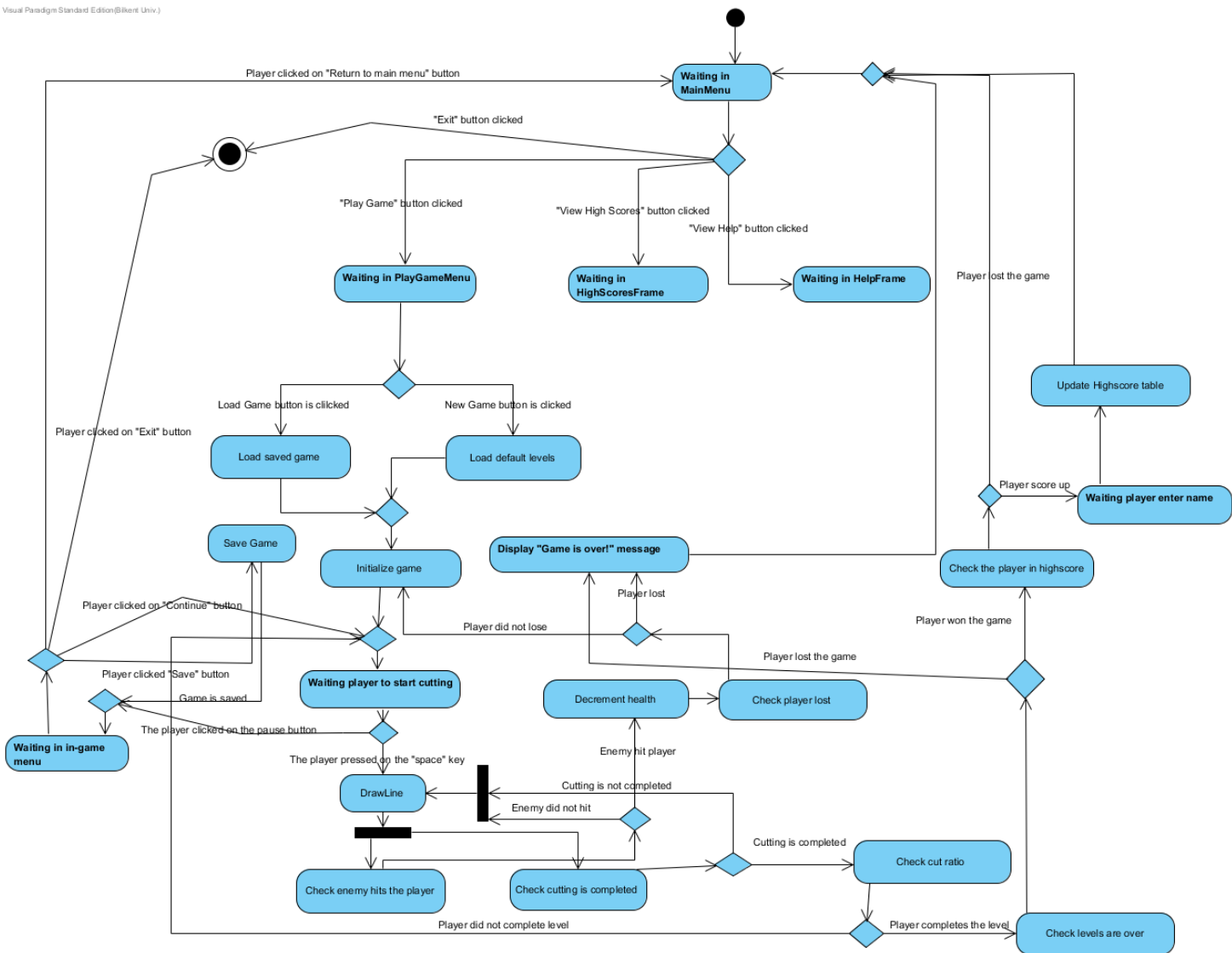


Figure 27. Activity Diagram

4. Conclusion

In this report, we introduce and analyzed our project the game “Split-Field” for CS319 course. Our report consists of the main parts which are requirements elicitation and system model design. The purpose of this analysis report is facilitating the implementation and design part of our project. With the guidance of this report, our implementation time and the possible bug number will decrease since we created possible scenarios that the player can role.

In the first part requirements elicitation, initially, we determined functional and non-functional requirements to identify the purpose of the system. Then we tried to examine all possible scenarios that player could perform during the lifecycle of the game which helped us to determine what is inside and what is outside of the system. After defining the system with requirements elicitation part, we started to determine system model design.

The second part of this report which is system model design consists of object models and dynamic models. This part contains the design of all classes and contents, methods and functions of these classes and the interactions and relations between these classes. The class diagram which contains classes, functions and relations between them will form the structure of implementation of our project, so it will reduce the possible problems that may occur because of lack of hierarchy and provide a way to reduce the complexity of the system by determining classes and their tasks.

Other than the class diagram, this report contains diagrams that visualize the design stage of the system such as use case diagrams, sequence diagrams, state chart diagrams and activity diagrams. These diagrams will help us to reduce the complexity of the system by separating the system to parts and analyzing these parts separately and also analyzing relations between these

separate parts. Thus this separate and conquer approach of diagrams will help us to understand the system effectively.

Finally, this report also contains the navigational path diagram and the user interface design of our project. We tried to keep the GUI of the project consistent and simple as possible to provide a usable and friendly interface that the player can have easy and familiar control on the game. While creating the navigational path diagram, we based on our use case diagram. Designing the user interface in this report will provide us to create a better user experience for the player as well as helping creation of distinction between lifecycles of the system.

To sum up, we acknowledged the fact that the analysis reports form the basis of projects by providing consistent structure for a solid software design. Thus, we tried to analyze this project as detailed as possible within this report with the expectation that this report will be guidelines for the implementation and design of the future steps of the project.

References:

[1]: <http://www.page-online.nl/volfied/index.php>

[2]: <http://www.page-online.nl/volfied/index.php?c=thegame>



Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering Project

CS 319 Project: Split-Field

Design Report

Ali İlteriş TABAK, Efe TUNÇGENÇ, Gizem ÇAYLAK

Course Instructor: Hüseyin Özgür TAN

Progress

Nov 20, 2015

This report is submitted to the Github in partial fulfillment of the requirements of the Object Oriented Software Engineering Project, course CS319.

Table of Contents

5.	Design	55
5.1.	Design Goals	56
5.1.1.	End User Criteria.....	56
5.1.2.	Maintenance Criteria.....	56
5.1.3.	Performance Criteria	57
5.1.4.	Trade-offs	58
5.2.	Sub-System Decomposition.....	59
5.2.1.	User Interface Subsystem Interface	62
5.2.2.	Game Management Subsystem Interface.....	63
5.2.3.	Game Entities Subsystem Interface.....	65
5.3.	Architectural Patterns	66
5.3.1.	Layers.....	66
5.3.2.	Model-View-Controller Architecture.....	67
5.4.	Hardware/Software Mapping	67
5.5.	Addressing Key Concerns	69
5.5.1.	Persistent Data Management.....	69
5.5.2.	Access Control and Security	69
5.5.3.	Global Software Control	70
5.5.4.	Boundary Conditions	70

5. Design

-Our progress so far:

In our previous report, we focused creating our game “Split-Field” just from scratch. By doing so we were able to determine necessary components of our project in order to complete our analysis report in the first place. After clarifying functional, non-functional and pseudo requirements needed for this project, we forwarded ahead with schematics. With drawing different kinds of use case models and also putting them into the words in report we were able to advance through our drawing step: class, sequence and activity diagrams are all drawn through these steps. Also we sketched some graphical character design and user interfaces, which we expect to see in our final demo.

Things that are going to be disclosed in this report will help us guide through our whole implementation stage. Starting with defining our design goals, we will progress further with subsystem decomposition, architectural patterns, hardware software mapping, persistent data management, access control and security and boundary conditions. In light of these topics, our design pattern will be selected.

5.1. Design Goals

Split-Field aims to be a user friendly game based on good timing and enemy tracking skills. In order to accomplish that, some different techniques must be achieved in order to reach our design goals. Overall, we want our game to have reliability, efficiency and ease of implementation. Design goals could be considered of continuation of non-functional requirements from our earlier report.

5.1.1. End User Criteria

- Clarity: While interacting with end users, being specific and unambiguous is an absolute necessity since there is no third person to rely on communication. Also, if a case occurs which an end user is not able to comprehend the navigation, or gameplay controls in the game, there is always an available help menu is option in main menu.
- Easy to play: Our game welcomes every generation. Since it must identify itself as “playable” to all generations, its difficulty level must be scaled into an amount that it is not very easy and not too much hard at the same time. Also, controls of our game does not include too much of control keys. Thus, it holds no difficulty with controls either.

5.1.2. Maintenance Criteria

- Adaptability: The game should not show any trouble on running different environments and platforms since it is going to be written in Java language. Only requirement for a computer to run our program is to have Java Runtime Environment loaded in itself. Otherwise, it should not be a problem.

- Extensibility: Extensibility provides such freedom to a software developer that they can think about what comes first in a project and maintain their focus in the main core of the mechanism. After putting the system on its track, additional classes like “new enemies” or “new levels” could be implemented in project. Using an object-oriented programming language gives this opportunity to add new classes without thinking about other ones. With this and low coupling, system becomes much more extensible.
- Modifiability: As told above, low coupling makes it possible for us to change and modify our classes, if there is one needed in our further steps in implementation.

5.1.3. Performance Criteria

- High Frame Rate (Smoothness)/Low Response Time: Any players worst nightmare is a freezing or lagging play screen in front of his/her. In order to avoid this, we are going to design our game engine to be fully functioning and giving 25 frames at minimum. Thus, a normal human eye could not detect any frame skip.
- Efficiency: Java is not famous with its best performance. So our system is going to work in the most efficient way possible to overcome downsides of Java environment.
- Reliability: Game should neither crash nor give any kind of error in its runtime. Split-field will be tested with various numbers of inputs to prevent any case like this. Also, system is going to be designed such a way that wrong inputs that user enters would not be accepted to our game.

5.1.4. Trade-offs

- Easy to use vs. Functionality: Making a game simpler means it has lesser functions inside it. More options a game has, more complex it gets. In this trade-off, we favored into simplicity because our original root game is a really simple, simple objective required game. And also, as told previously above, our game is intended for a wider user range including different generations.
- Performance vs. Resource Consumption: Split-field is a fast and reflex-based game. In order to reflect this idea, performance should be without compromise. Memory will be used in the most efficient way to make our compromise minimum. Besides, our game is not a system tiring program and if we take consider of memory capacity of nowadays computers, users will not notice the difference between a super-fast algorithm and a fast algorithm.
- Efficiency vs. Reusability: If we were some entrepreneurs in gaming industry and thinking about the possibility of publishing a sequel to this game, reusable source code could be really beneficial for us to build similar mechanism and environments. But here we are as students, aiming to give a fully working demo. So, instead of focusing on reusability, we are going to try to make it more efficiently.

5.2. Sub-System Decomposition

In this section, we decomposed our system into smaller parts based on uses cases and analysis models of the system. To obtain ideal decomposition, we determined the main purpose of this decomposition as reducing the coupling between various subsystems as well as increasing cohesion within subsystems. As a starting point during this activity of decomposition, we used Model/View/Controller (MVC) architectural style which will make our implementation more maintainable and reusable. Decomposition of the system into relatively independent subsystems to demonstrate organization of the system clearly is crucial for meeting non-functional requirements as well as creating high quality implementation by retaining the significant features of our software system like extendibility, reusability, performance.

Initially, we addressed system-wide issues and determined three main subsystem that can be realized independently and encapsulates the states and behaviours of its contained class. Thus as in Figure 28, we decomposed our system into three subsystems: User Interface, Game Management and Game Entities.

The first subsystem “User Interface” provides user interface components for displaying to the user and represents the view part of MVC architecture. The second subsystem “Game Management” is responsible for control and manages the user input and game, and represents the control part of MVC architecture. The third subsystem “Game Entities” contains the entities and game objects, and represents the model part of MVC architecture.

Figure 29 demonstrates the connections between subsystems which based on low coupling between subsystems and high cohesion within subsystems. As seen from figure, the only connection between “Game Management” and “Game Entities” subsystems provided over

“GameObject” class. Owing to this loosely coupling connection, any change on “Game Managment” subsystem will only affect “GameObject” class rather changing whole “Game Entities” subsystem. From figure 29, when the subsystems are examined separately, we can conclude that high cohesion within subsystems provided by putting together the classes which perform similar tasks and have similar purposes within the system to same subsystem.

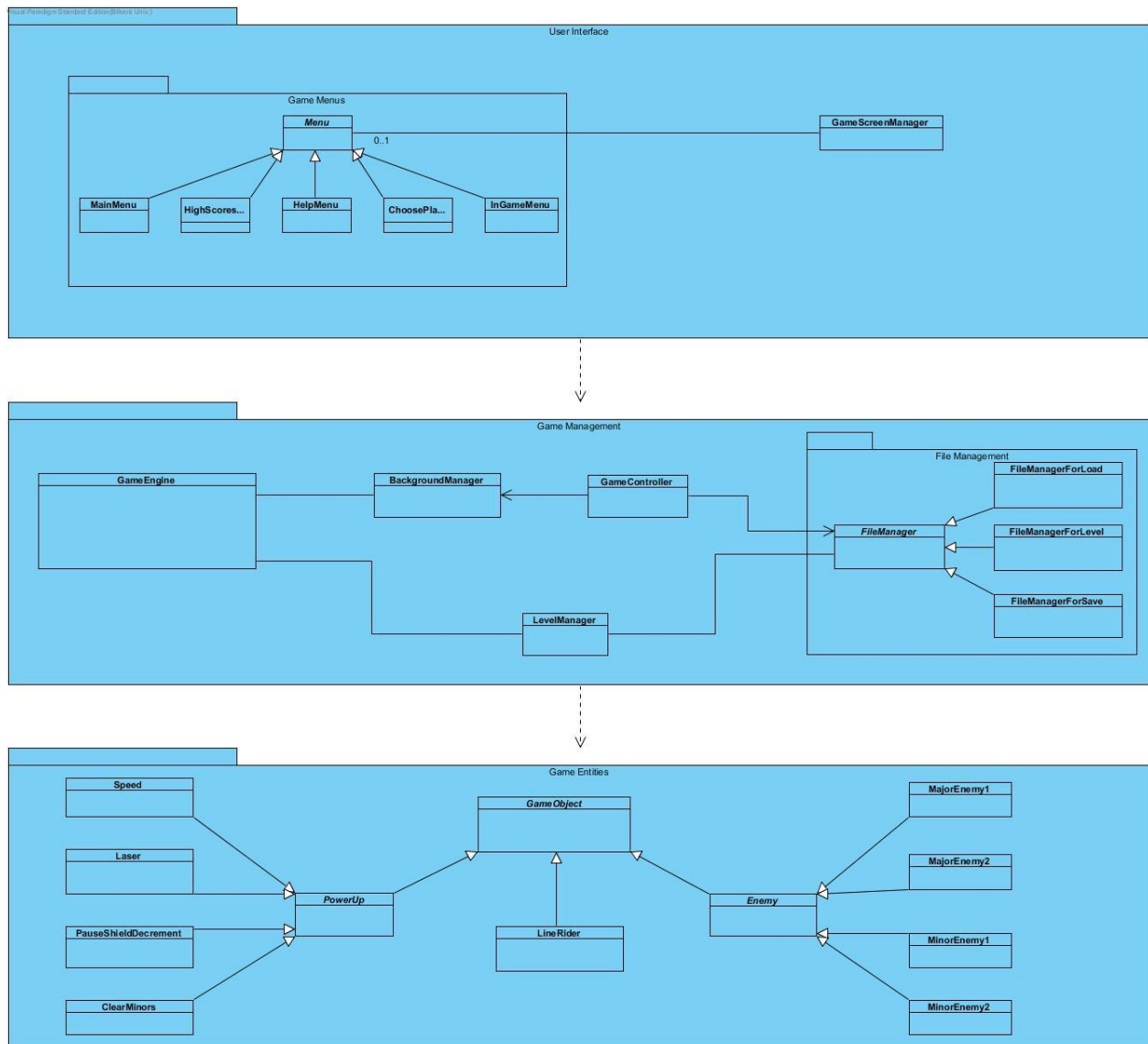


Figure 28. Basic Subsystem Decomposition

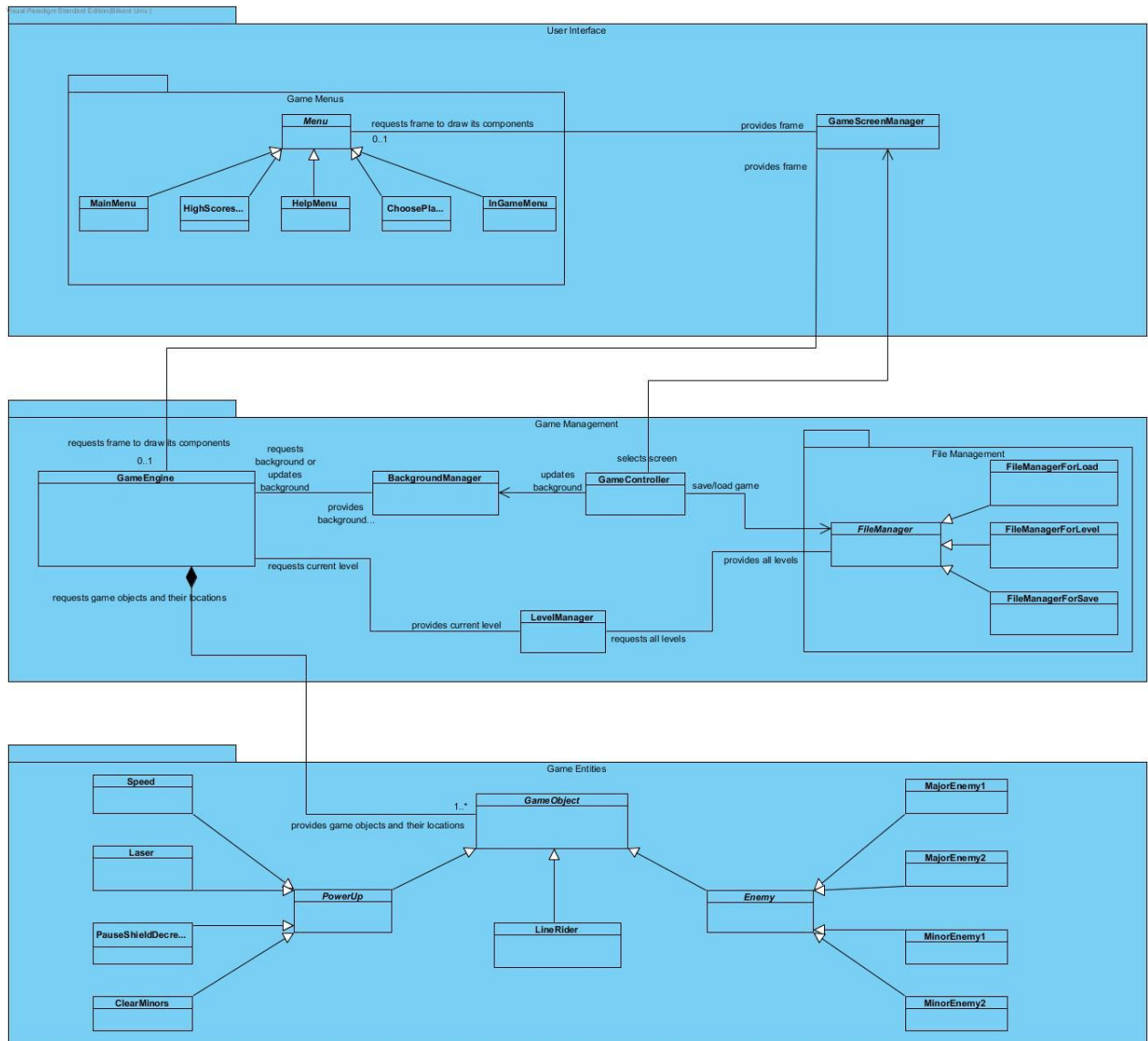


Figure 29. Detailed Subsystem Decomposition

5.2.1. User Interface Subsystem Interface

User Interface Subsystem provides the static stable interface components (Game Menus) and a management component for the user interface to our system. As shown below, the subsystem is composed of a Game Menu Subsystem and a GameScreenManager Class. Game Menu Subsystem contains 6 classes one of which is abstract class, Menu that is extended by other 5 classes which can be considered as the Views of the MVC architecture. Each class, that extends the Menu abstract class, contains the required instances for its purpose. GameScreenManager is, as understood from its name, the manager for user interface components. It holds the object that creates current user interface. Constructor of GameScreenManager takes a MainMenu object as its parameter because the first user interface component seen by the user is MainMenu. With the proper mouse inputs, GameScreenManager changes the objects that create the user interfaces. Details of services that are provided by aforementioned classes can be examined in [Analysis Report](#). The reference of User Interface Subsystem to other subsystems is provided by GameScreenManager class.

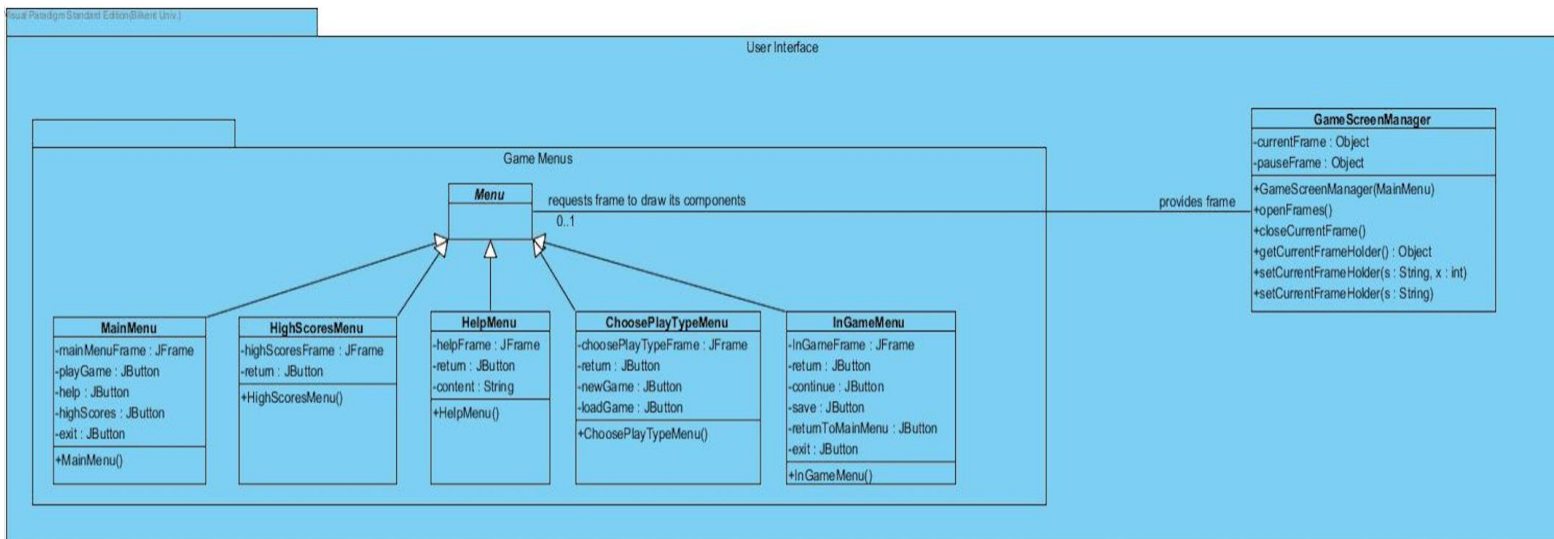


Figure 30. User Interface Subsystem

5.2.2. Game Management Subsystem Interface

Game Management Subsystem provides controller components of our system to manage the game dynamics and game logic, additionally inputs of user. As shown in figure 31, the subsystem is composed of GameEngine class, BackgroundManager class, GameController class, LevelManager class and File Management Subsystem. File Management Subsystem contains 4 classes one of which is an abstract class, FileManager, that is extended by other 3 classes (FileManagerForLoad, FileManagerForLevel, FileManagerForSave). Each class, that extends the FileManager abstract class, contains the required instances for its purpose. GameController component, which can be considered as the Controller of the MVC architecture, controls the flow of system with the inputs taken from the user. It manages the object that creates current user interface through the GameScreenManager component of User Interface Subsystem when user is in one of the menus of the system or it updates the BackgroundManager instance according to the keyboard inputs taken from the user. Additionally, it manages the save and load game options of the system. BackgroundManager component, which can be considered as the Model of MVC architecture, helps GameEngine component by managing the state of the cut field and other dynamics of game, like checking the properness of cut. GameEngine component draws the dynamic game field with the information taken from the game objects and BackgroundManager component. Through the flow of events in the gameplay, it updates the game objects and BackgroundManager object. LevelManager component helps GameEngine with acquiring information of all levels using File Management Subsystem and maintaining these levels during the execution of the system. Details of services that are provided by aforementioned classes can be examined in [Analysis Report](#).

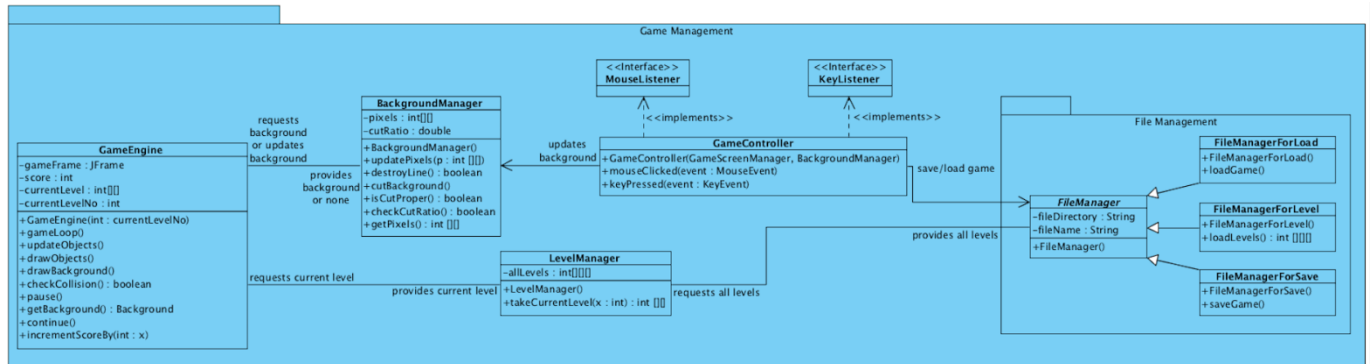


Figure 31. Game Management Subsystem

5.2.3. Game Entities Subsystem Interface

Game Entities Subsystem provides domain specific components of our system. As shown in figure 32, the subsystem is composed of 12 classes one of which is an abstract class, `GameObject` that is extended by `LineRider` class, `PowerUp` abstract class and `Enemy` abstract class. Existing components in this subsystem can be considered as the Model of MVC architectural style. Details of services that are provided by aforementioned classes can be examined in [Analysis Report](#).

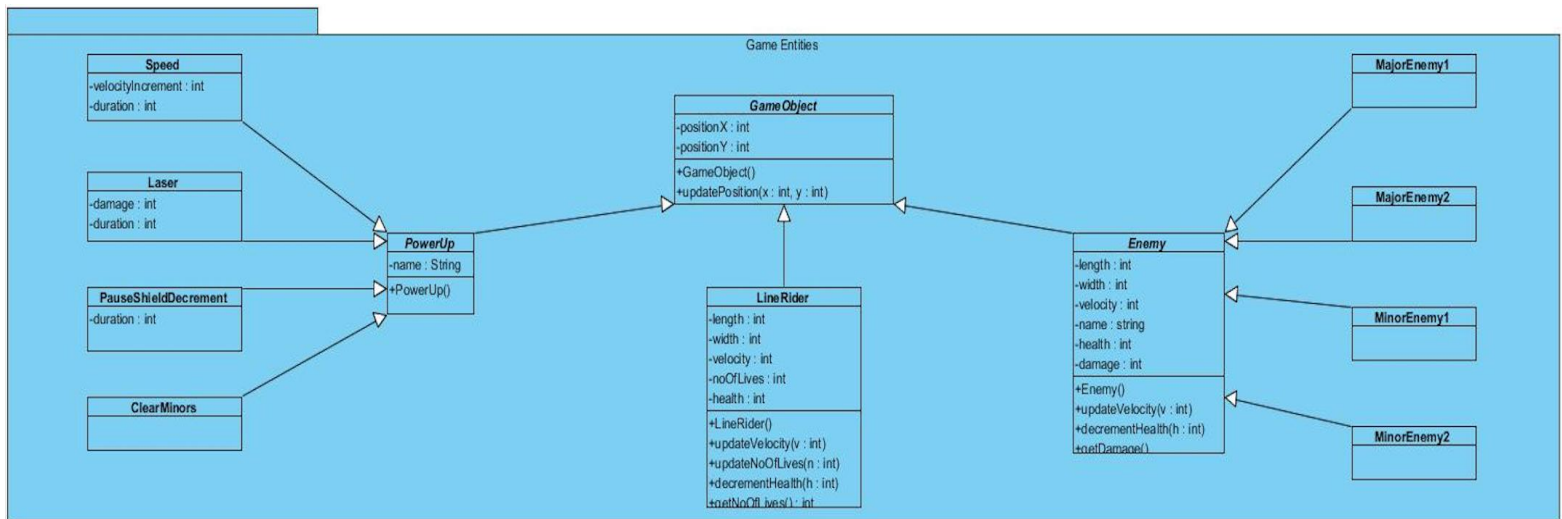


Figure 32. Game Entities Subsystem

5.3. Architectural Patterns

5.3.1. Layers

When decomposing our system, we used hierarchical decomposition which yields an ordered set of layers. Our system consists of three hierarchical layers which are the top layer “User Interface”, the middle layer “Game Management” and the bottom layer “Game Entities”. The top layer “User Interface”, which has the knowledge of the layers below it, provides user interface components for displaying to the user and represents the view part of MVC architecture. The middle layer “Game Management”, which depends on “Game Entity” layer and has no knowledge of the top layer “User Interface”, is responsible for control and manage the user input and game, and represents the control part of MVC architecture. The bottom layer “Game Entity”, which does not depend on any other layer, contains the entities and game objects, and represents the model part of MVC architecture. Our layer architecture also supports the closed architecture in which each layer can access only the layer immediately below it as seen in Figure 33.

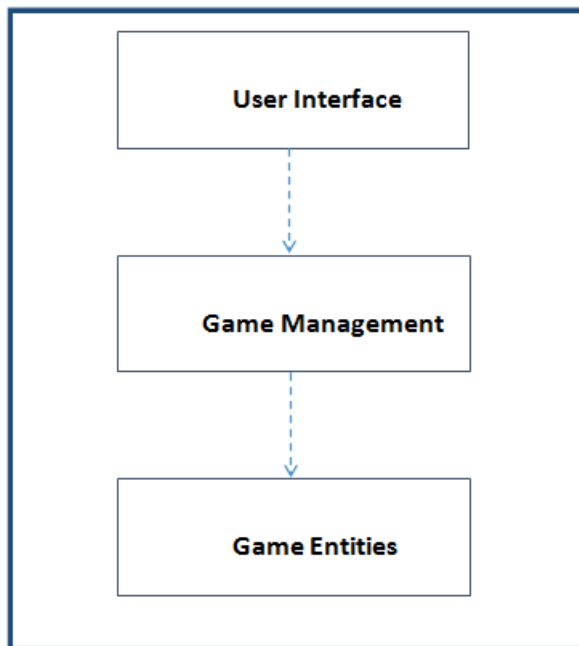


Figure 33. Hierarchical layers of system

5.3.2. Model-View-Controller Architecture

The decomposition of our system into subsystems is made accordingly to Model-View-Controller architectural style which consists of a model subsystem to maintain domain knowledge, a view subsystem to display the models to user, and a controller subsystem to manage the sequence of interactions with the user. With this architectural style, changes made by user via View subsystem, which corresponds to User Interface Subsystem in our system, is controlled by the Controller subsystem that corresponds to Game Management Subsystem in our system, before Controller updates the state of Model subsystem, which corresponds to Game Entities Subsystem in our system. The reason behind choosing this architectural style is its ease of extendibility on model components. In addition, the architectural style provides an independency between subsystems which will make implementation stage easier if further change or enhancement is required in any subsystem.

5.4. Hardware/Software Mapping

Split-Field will be implemented in Java programming language which will allow the game to be played platform independently. Therefore we will use latest JDK which is 1.8 currently for the implementation. The hardware configuration that the game requires is a keyboard and mouse. Keyboard will be used for two main functions: the first one is for enabling the user to enter his/her name to the high score table and the second is for let the user control the game with arrow keys to navigate the game field as well as start cutting with special keys such as space key. Mouse will be needed for users to click on buttons to navigate between screens. In addition to platform dependency advantage of java, the system requirements will be minimal and a standard

computer with an operating system installed a java compiler with it will run “*.java” file which is the format of our game.

Storage issue of our game will be solved by using the I/O libraries of Java that will enable us to store data in .txt files locally on user’s computer. With an algorithm that translates the progress of a player into proper text format will be saved into a “*.txt” file and the users progress will be read from same saved .txt file to load the game by using a reverse algorithm. High scores table will be stored in a .txt file and read from same .txt file as well. Since local storage is used management of data, to play the game, any kind of internet connection will not be needed.

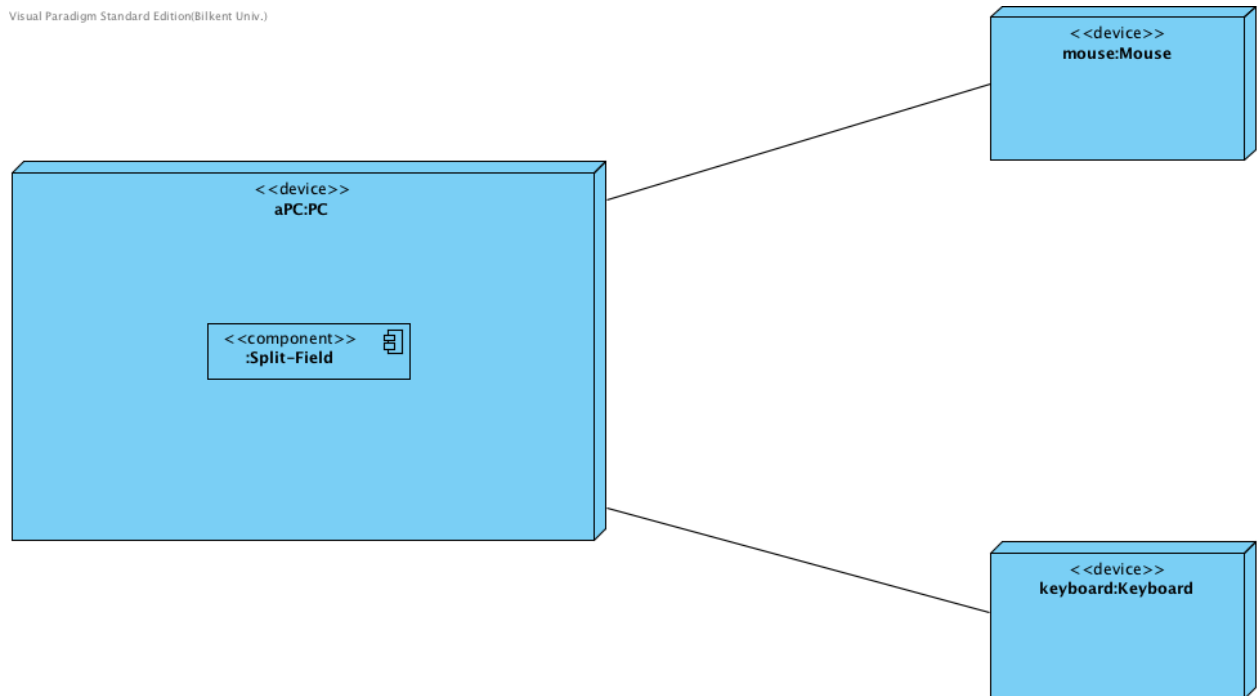


Figure 34. Deployment diagram of Split-Field

Nodes of the deployment diagrams are PC, Mouse and Keyboard as devices. Split-Field component provides services to a user. Because a component in a deployment diagram provides a higher-level of view for the component, Split-Field component is composed of 3 components which will be specified in details with their run-time relationships in Figure 35.

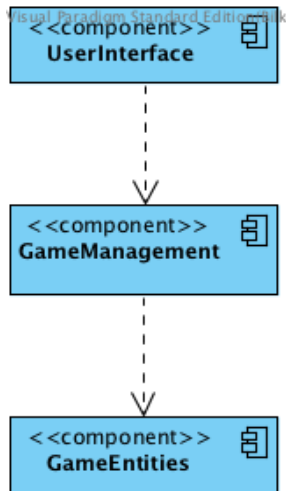


Figure 35. Component diagram of Split-Field component in the Deployment Diagram

5.5. Addressing Key Concerns

5.5.1. Persistent Data Management

Game is designed to hold the minimum number of files outside the program in order to gain modifiability for further design changes. There are two components, which are held outside the system in a local directory where program could reach: first one is the level number that user saved his/her game state. Second is a .txt file that contains properties of all levels such as background image, file name of an enemy graphic, how many minor and major enemies are going to be in that level and so on. These two files above will work synchronously since a loading process needs to get previous saved game's level number, and use this for getting the specific data needed for building that level with right components.

5.5.2. Access Control and Security

Our game is a single player game and therefore, it doesn't have any access to a network. Unless it is given permission for a remote access from that computer, there is no chance for user's data to be leaked. In fact, our game does change only one value from computer, which is

the level number when user saves a game. Since it is considered within our system, we could fully assure our users that Split-Field possess no threat for personal security.

5.5.3. Global Software Control

After our brainstorm on the subject, we decided to go with a three-layered subsystem pattern. Our group adjusted these layers in order to reflect the Model-View-Controller pattern in our game. Main idea here is to isolate subsystems and constitute low coupling between them. Also, grouping similar functioned classes in a single subsystem creates high coherence in our game. Both high coherence and low coupling are essential towards a good design and together they assist our system for reducing its complexity while allowing making changes. It should be noted that User Interface-View- and Game Entities-Model- could communicate only through Game Management-Controller-. This is the desired pattern for our game since it applies high coherence and low coupling principles.

5.5.4. Boundary Conditions

- Initialization: Our game does not need any additional files or setup to execute. It could be initialized on any computer with a clicking a single .jar file. Computer only needs Java Runtime Environment installed in it.
- Termination: An end user has three choices to exit Split-Field. First one is to choose “Exit Game” from our main menu manually. There is also an “Exit Game” button in our pause menu screen. But there is a point, which should be remembered: if current game were not saved, all gameplay data would be lost since it not saved before program termination. Thirdly, user is able to click a GUI component, which is an exit button on program screen provided by Java.

- Errors: For any coding or design mistake, which could go wrong, there could be possible errors. If there is a mistake in coding, program is most likely to give a runtime error even if it is compiled correctly. But even after it manages to survive execution, there still might be some design flows that can cause logical errors in our program. To give an example: there could be some menu options missing or some game mechanisms could work differently than the purpose we had in our mind.