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

Figure Table

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, the 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.

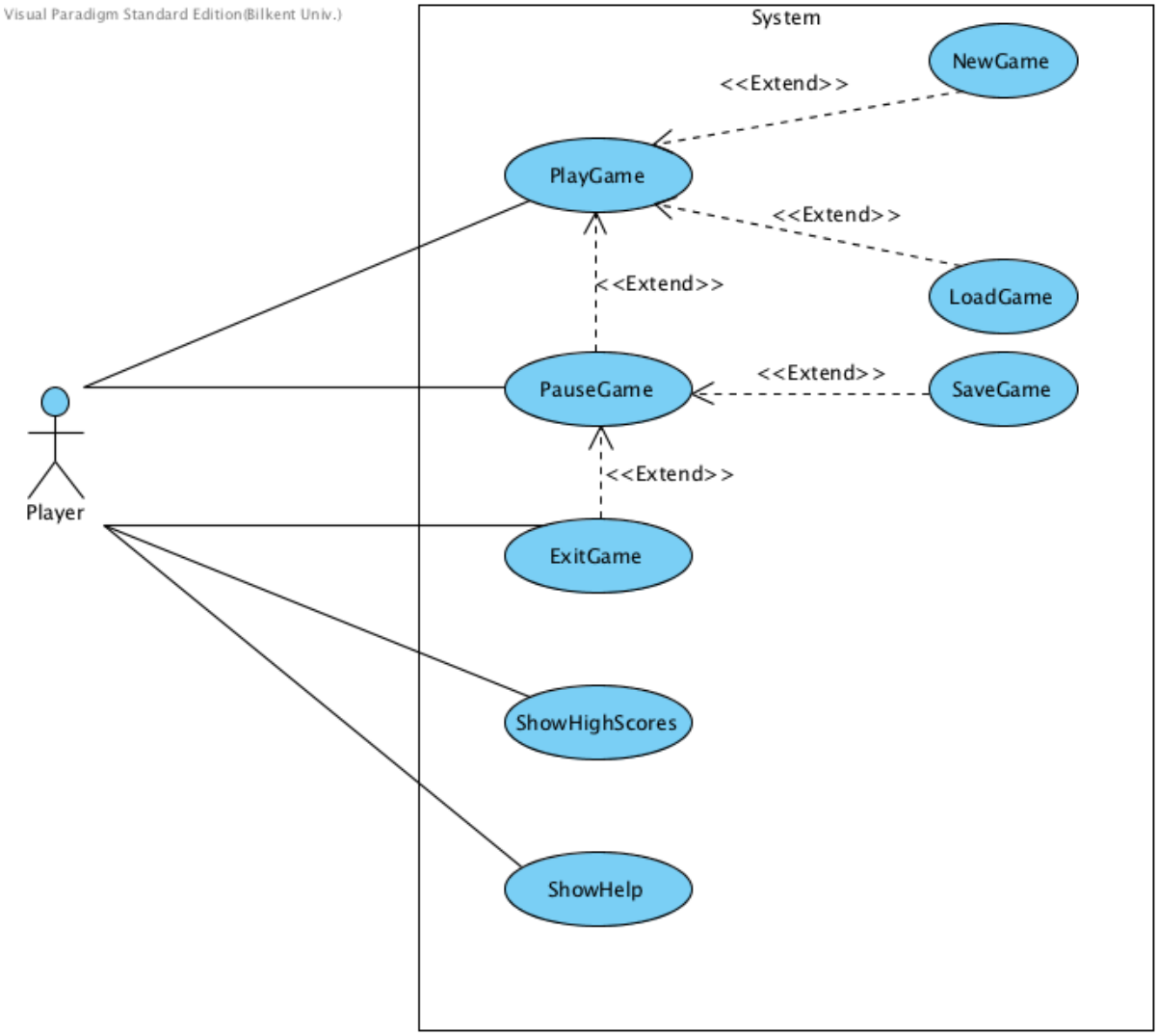


Figure 3. Use Case Diagram

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

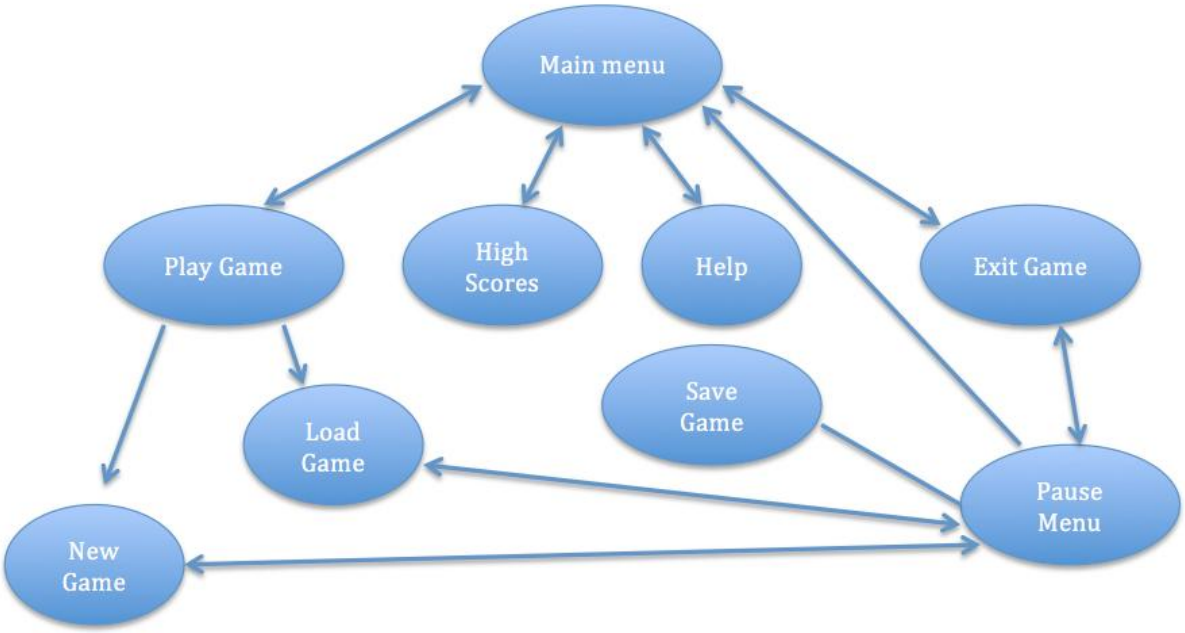


Figure 4. The navigational paths between the game menus

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

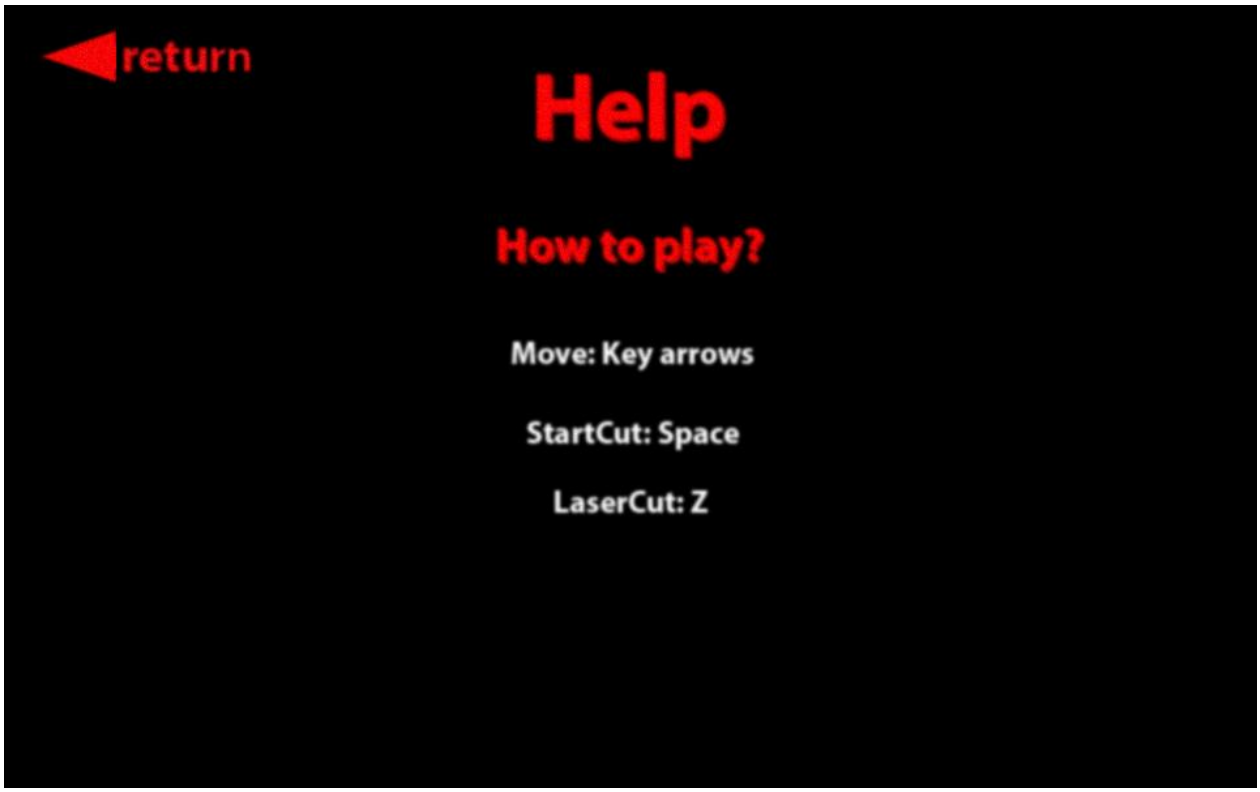


Figure 7 The mock-up of the help menu

2.7.2.4. Play Game Menu

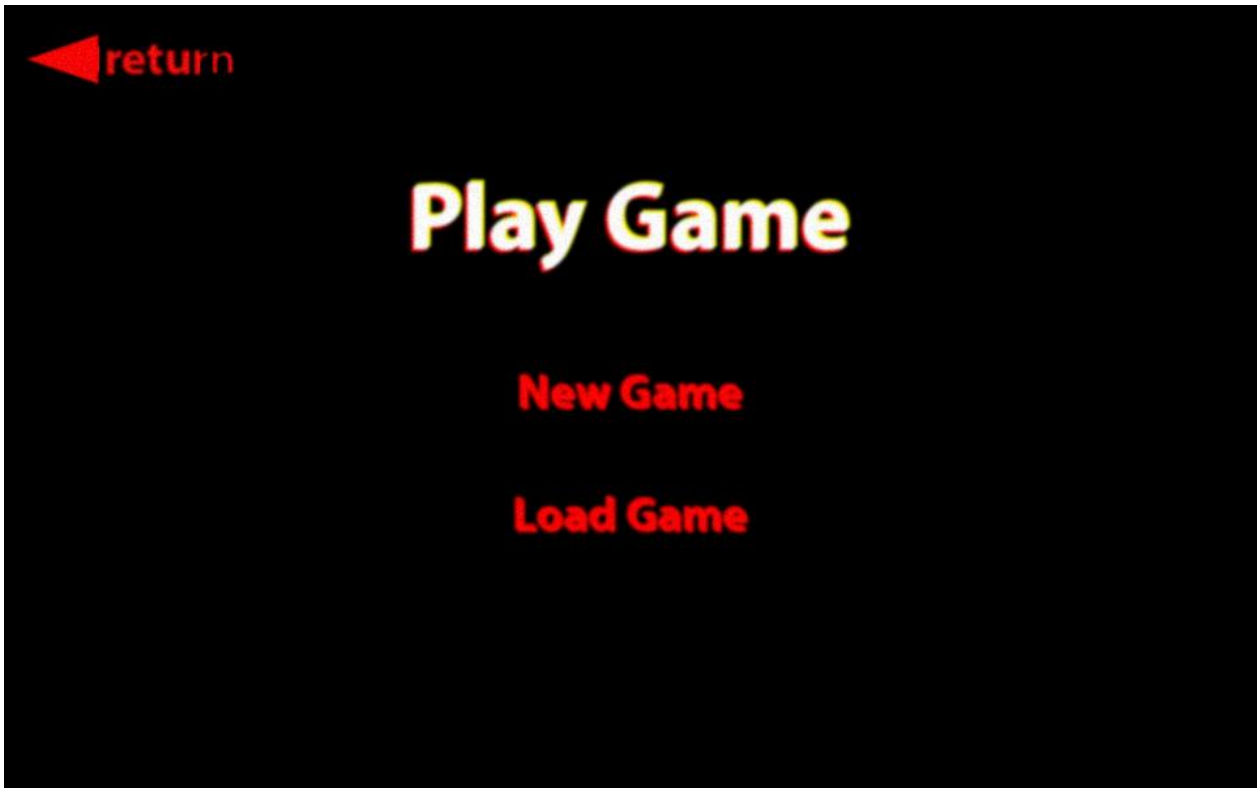


Figure 8 The mock-up of the play game menu

2.7.2.5. In-game Menu

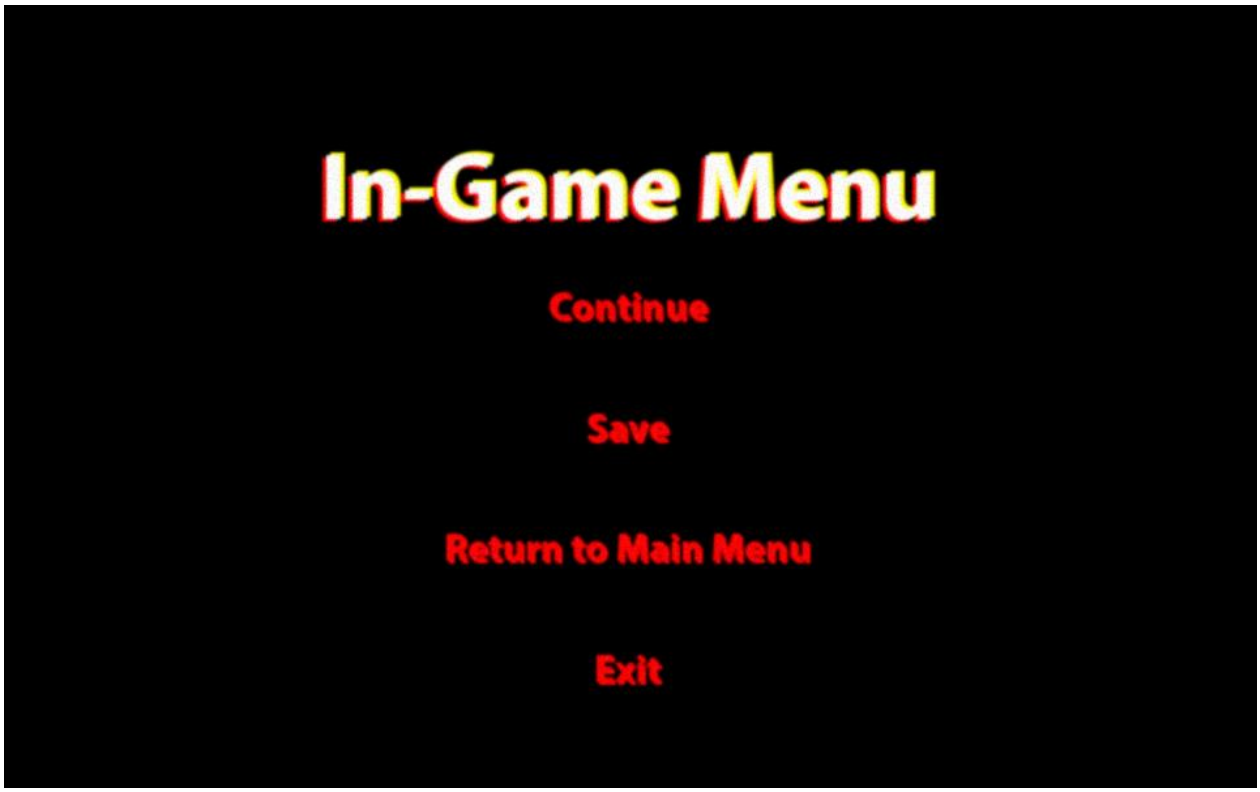


Figure 9 The mock-up of the play game menu

2.7.3. In Game Graphics

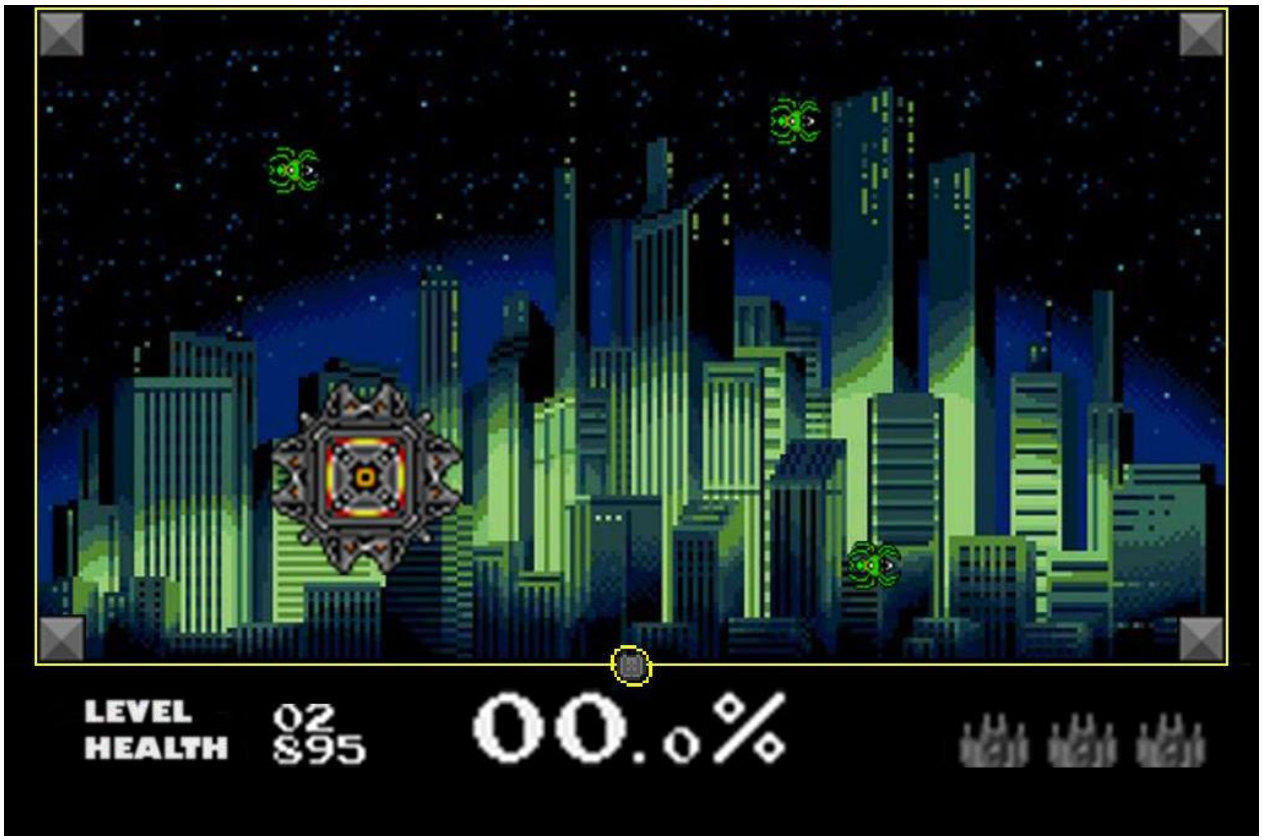


Figure 10. The mock-up of the game field

2.7.4. Characters

2.7.4.1. Line-Rider

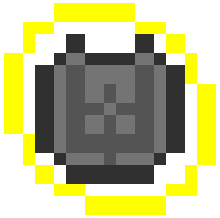


Figure 11. LineRider

2.7.4.2. Enemies

- Level 1 Major Enemy

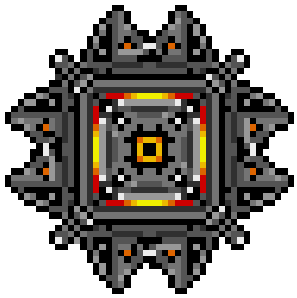


Figure 12. Level 1 Major Enemy

- Level 1 Minor Enemy

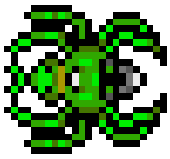


Figure 13 Level 1 Minor Enemy

- Level 2 Major Enemy

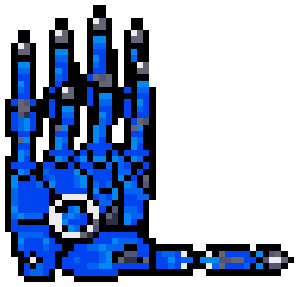


Figure 14. Level 2 Major Enemy

- Level 2 Minor Enemy

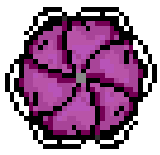


Figure 15. Level 2 Minor Enemy

- Level 3 Major Enemy

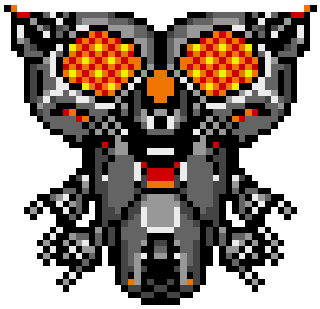


Figure 16 Level 3 Major Enemy

- Level 3 Minor Enemy

Figure 17. Level 3 Minor Enemy

- Level 4 Major Enemy

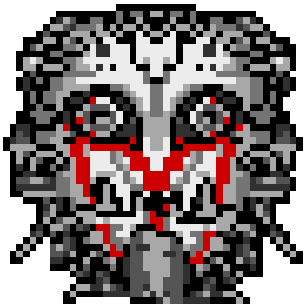


Figure 18. Level 4 Major Enemy

- Level 4 Minor Enemy

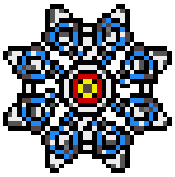


Figure 19. Level 4 Minor Enemy

- Level 5 Major Enemy

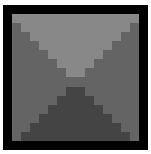
Figure 20. Level 5 Major Enemy

- 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

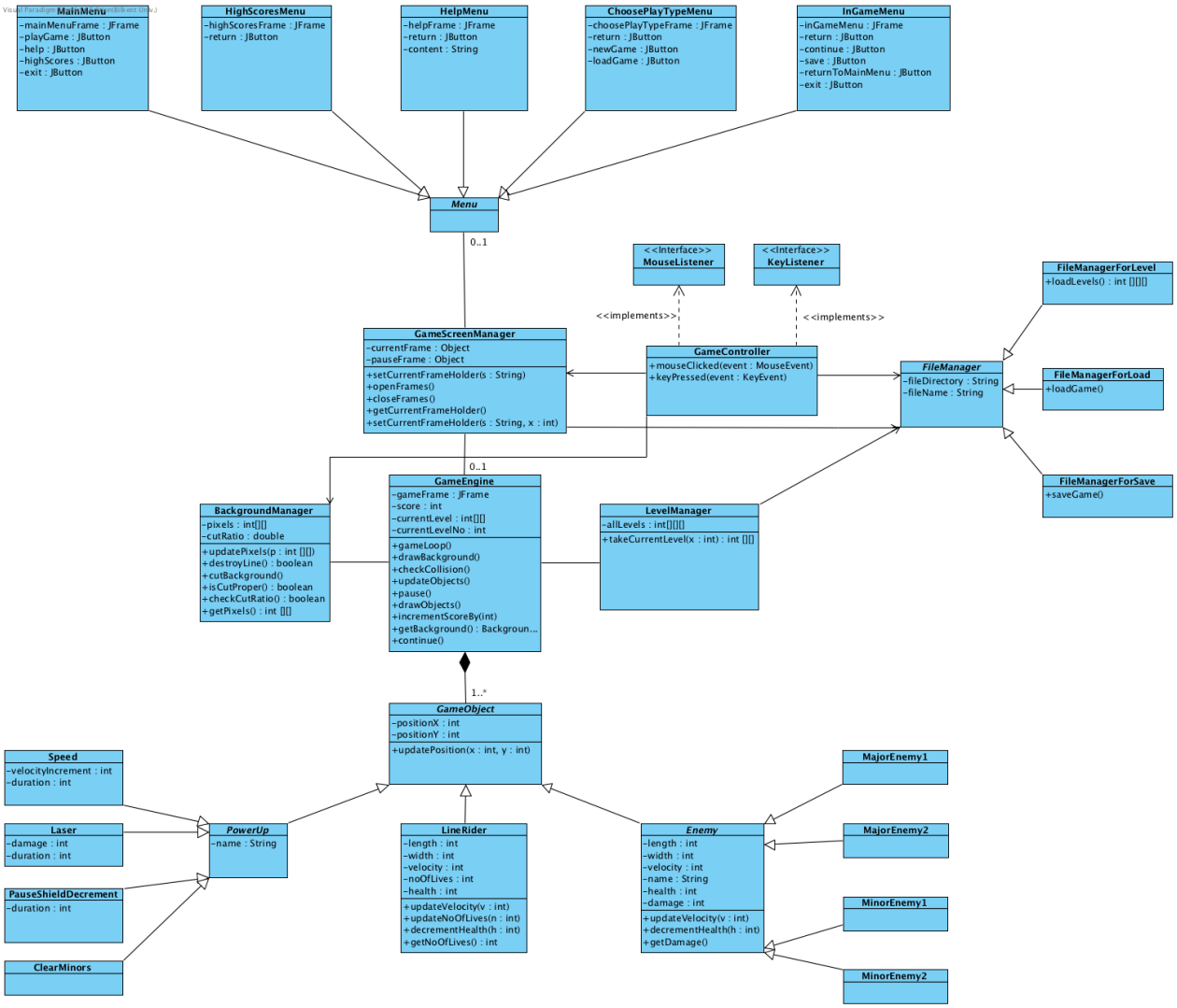


Figure 22. Class Diagram

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

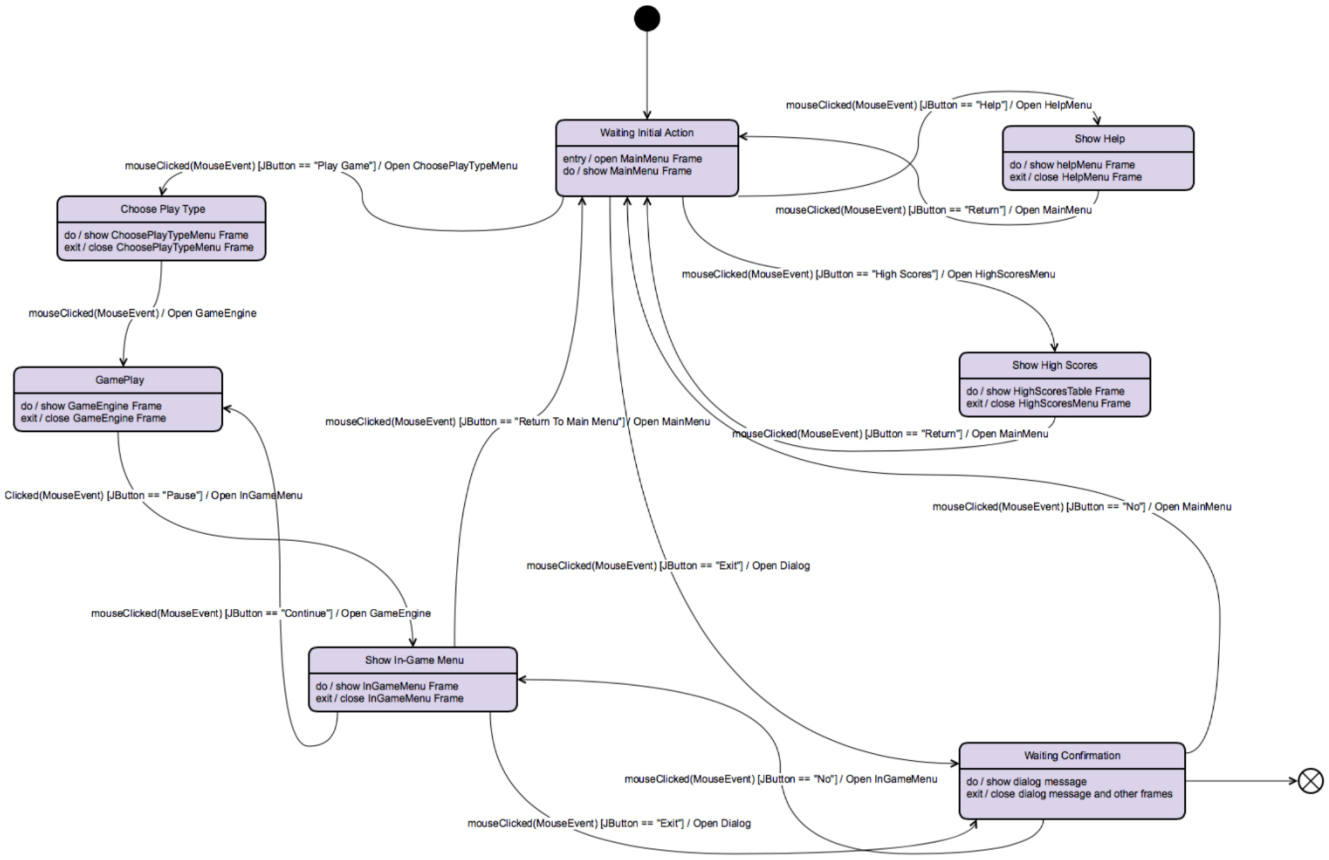


Figure 23. State chart diagram of GameScreenManager object.

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 24.

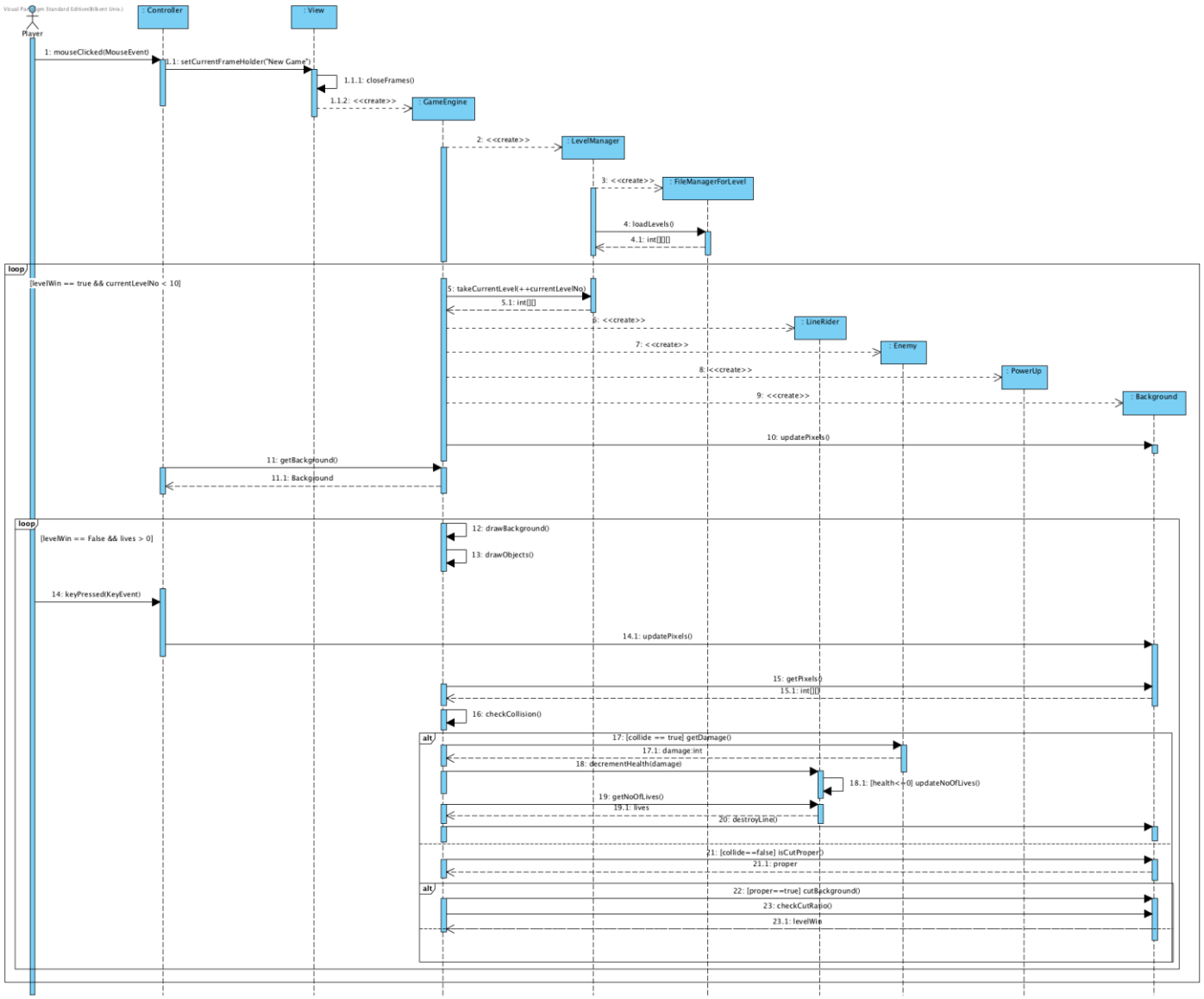


Figure 24. New game sequence diagram

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 25.

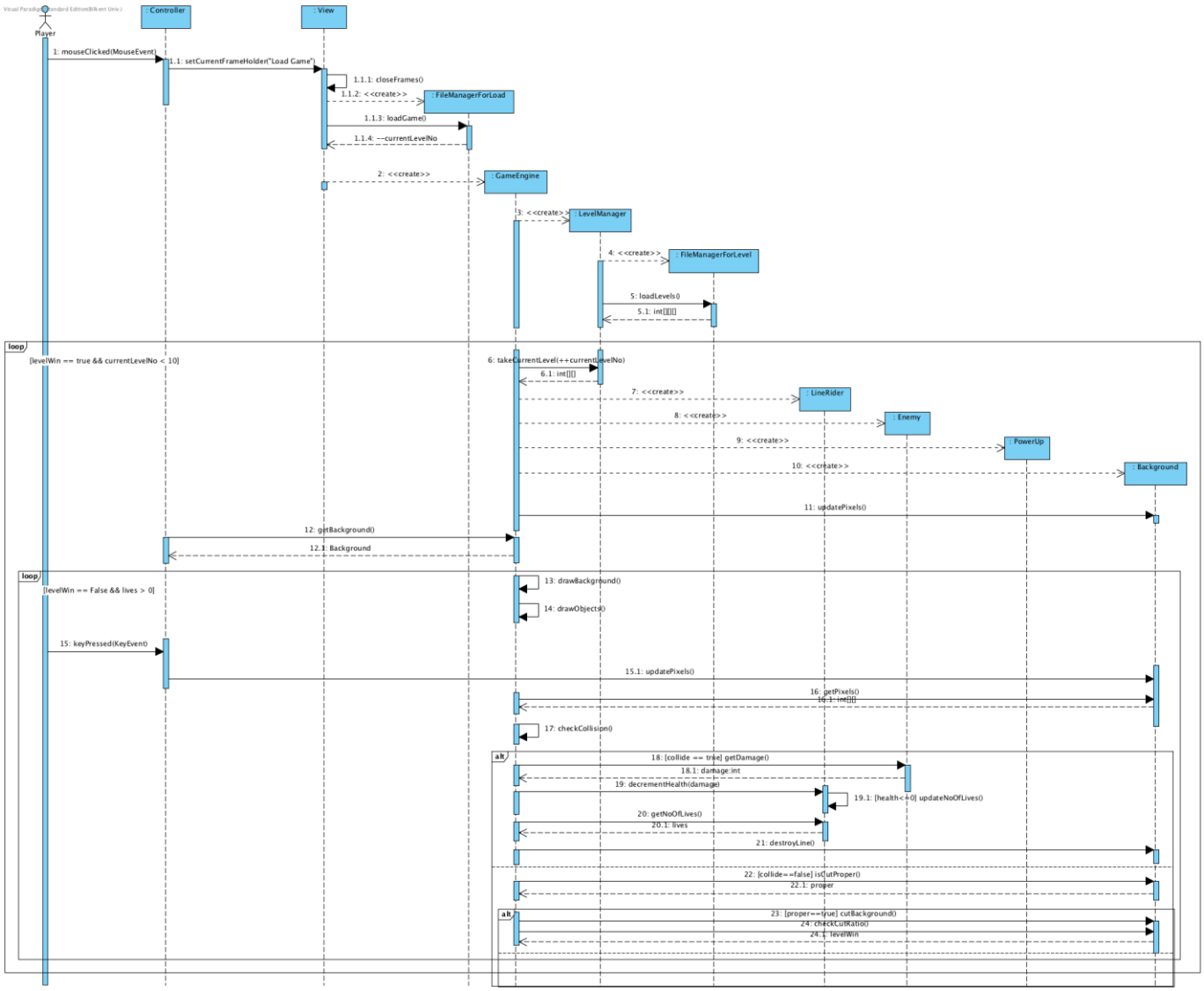


Figure 25. Load game sequence diagram

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 26.

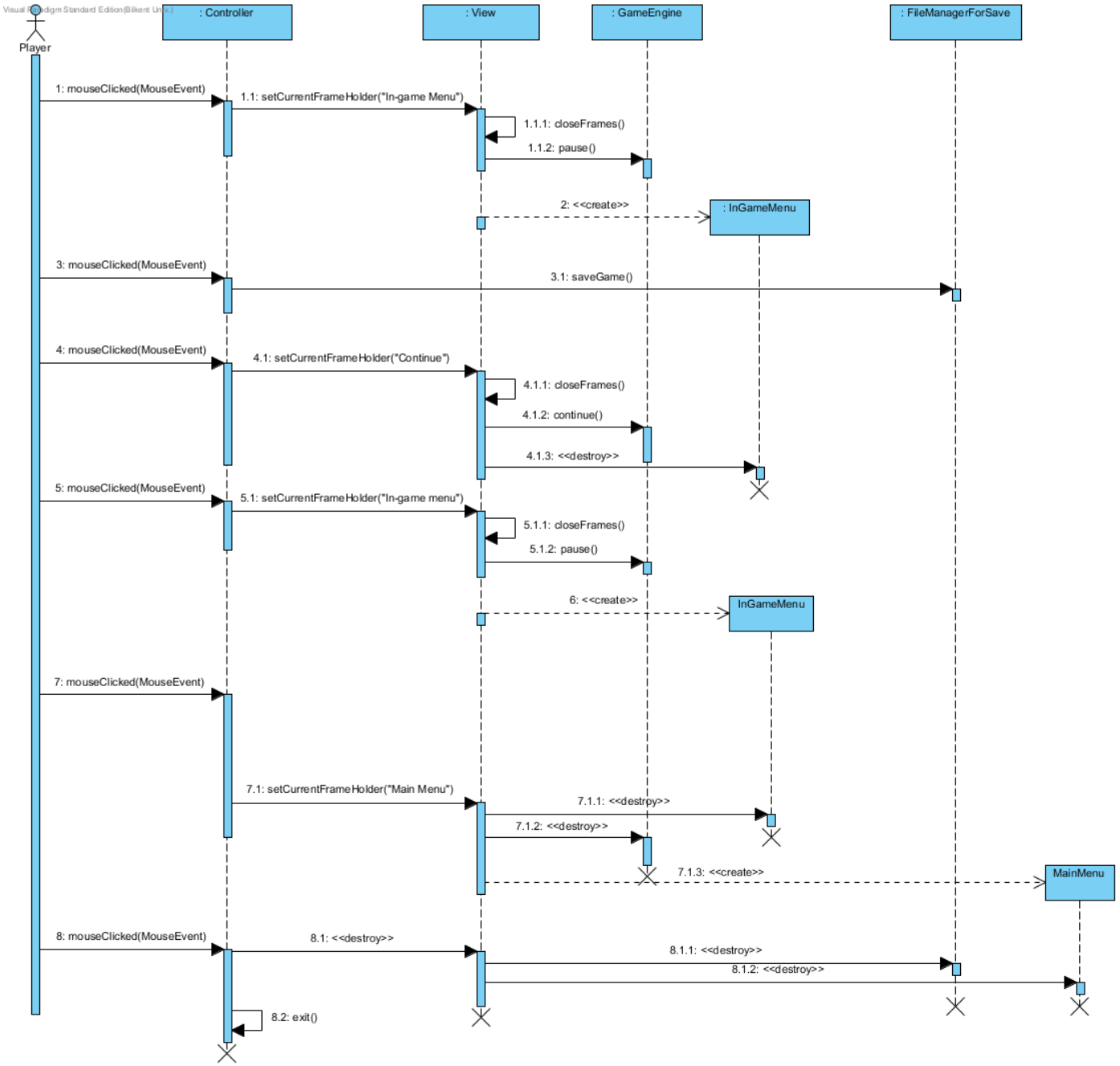


Figure 26. Pause & Save game sequence diagram

3.2.3. Activity Diagram

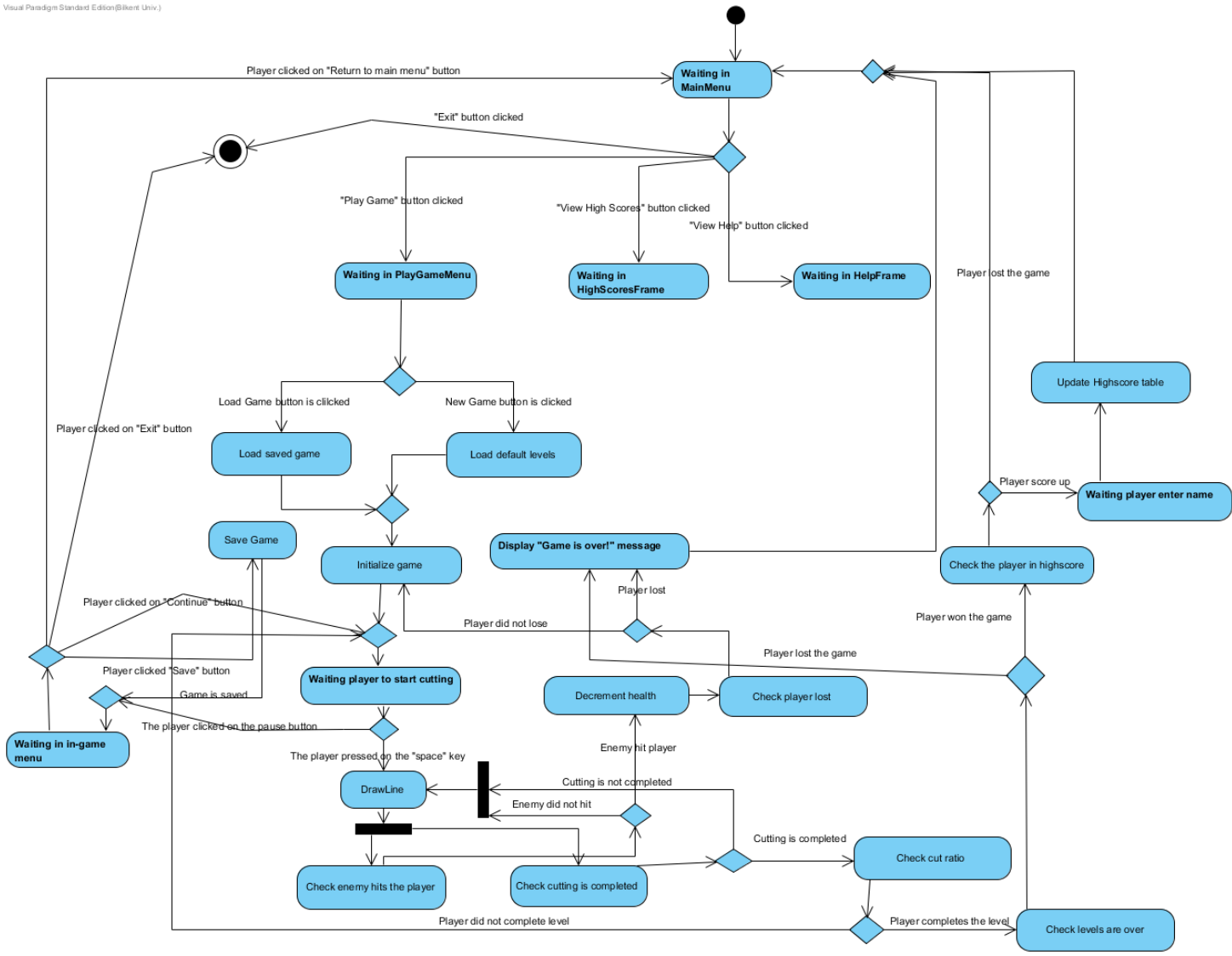


Figure 27. Activity Diagram

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

-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 Managament” and “Game Entities” subsystems provided over

"GameObject" class. Owing to this loosely coupling connection, any change on "Game Managament" 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.

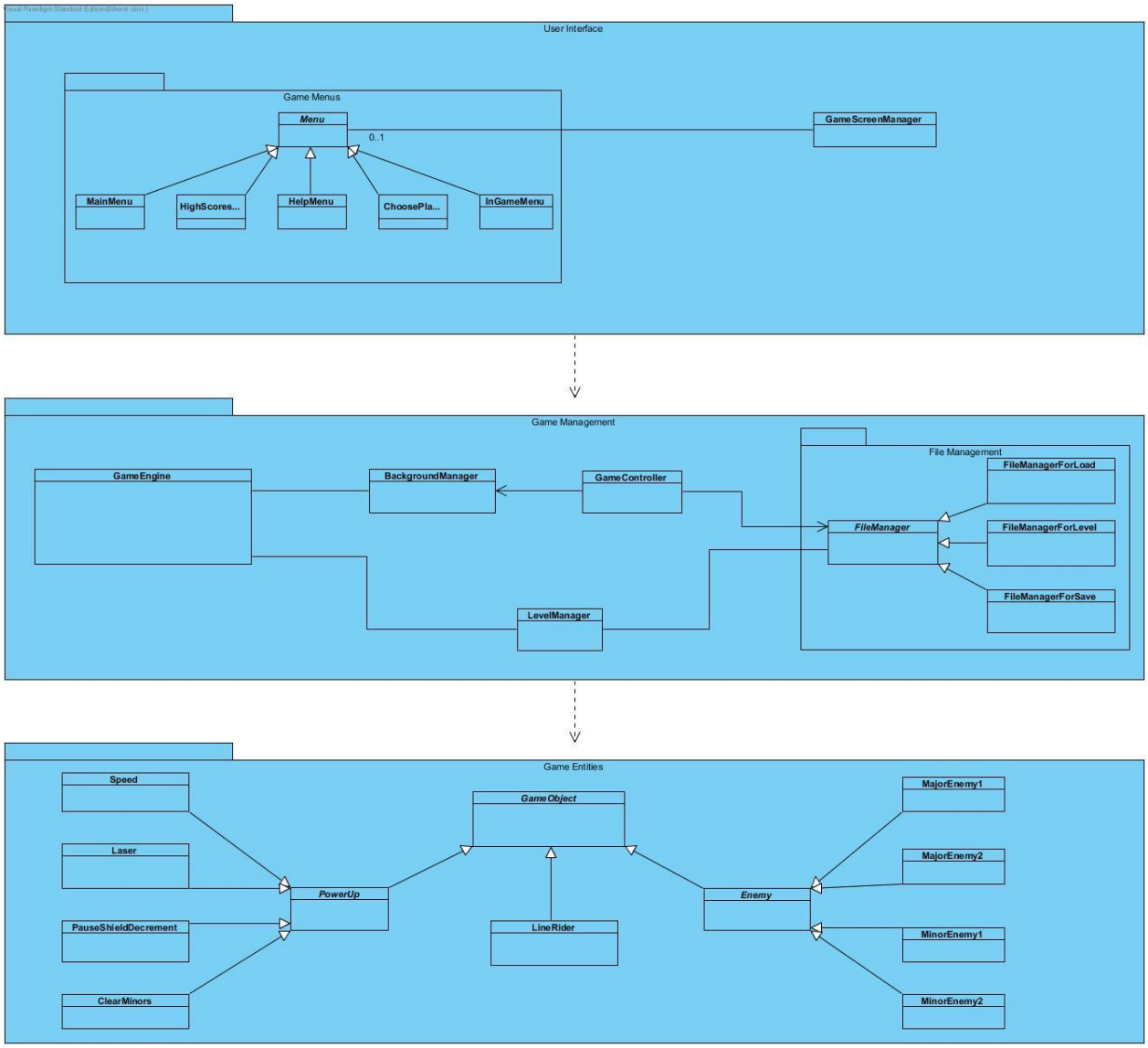


Figure 28. Basic Subsystem Decomposition

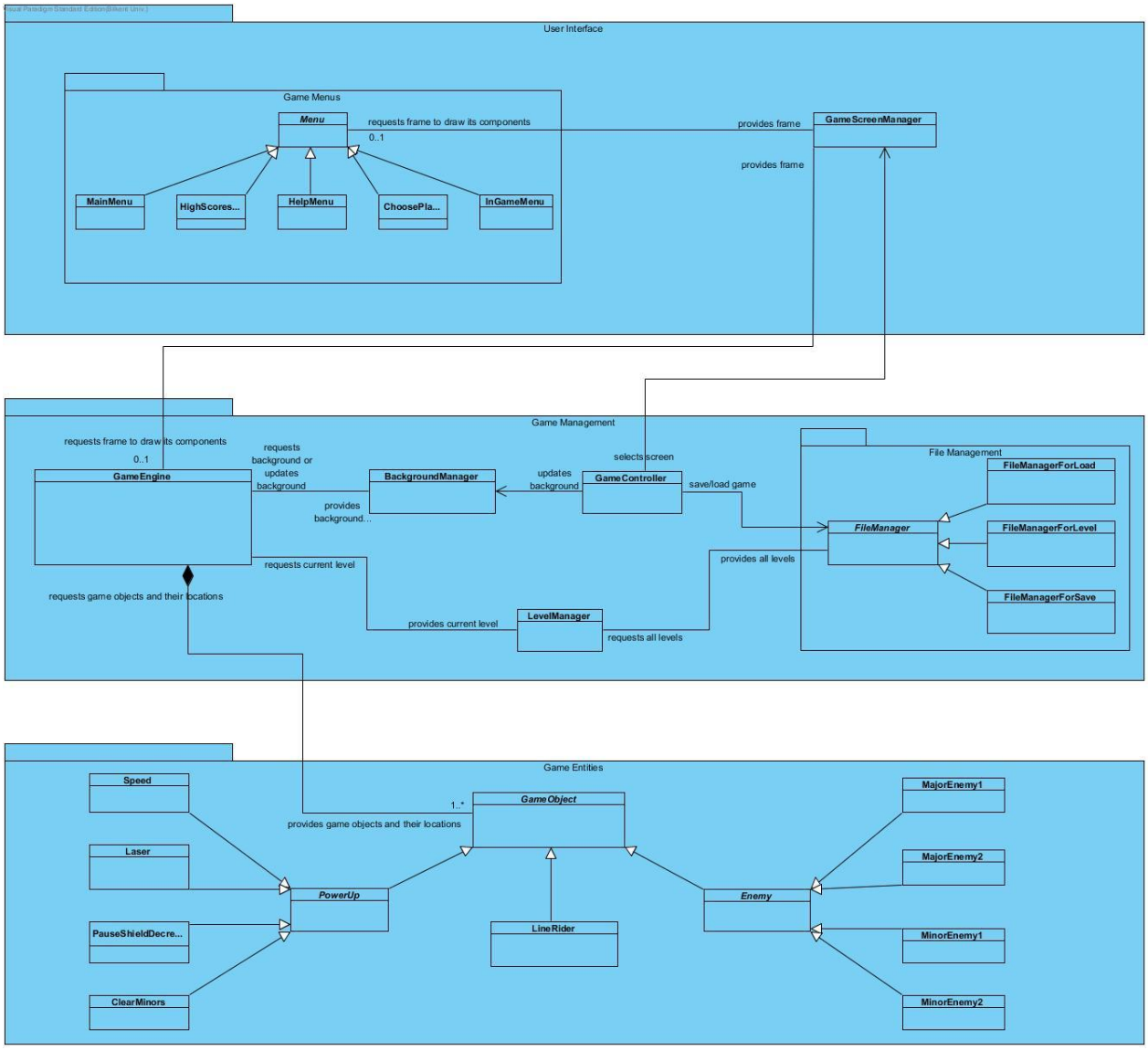


Figure 29. Detailed Subsystem Decomposition

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 Menus 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.

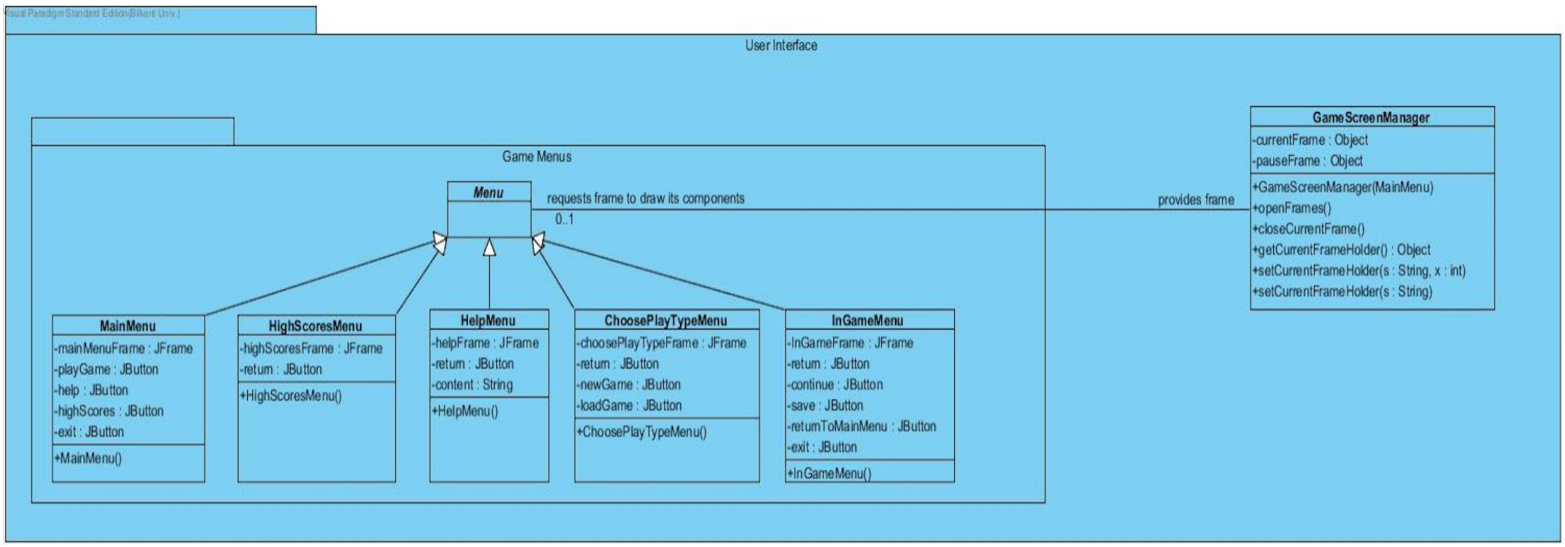


Figure 30. User Interface Subsystem

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 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.

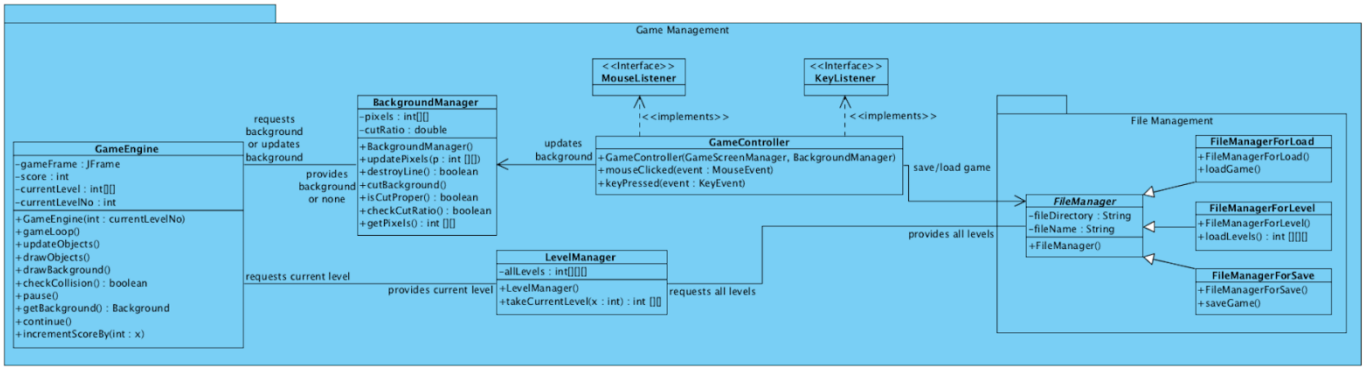


Figure 31. Game Management Subsystem

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.

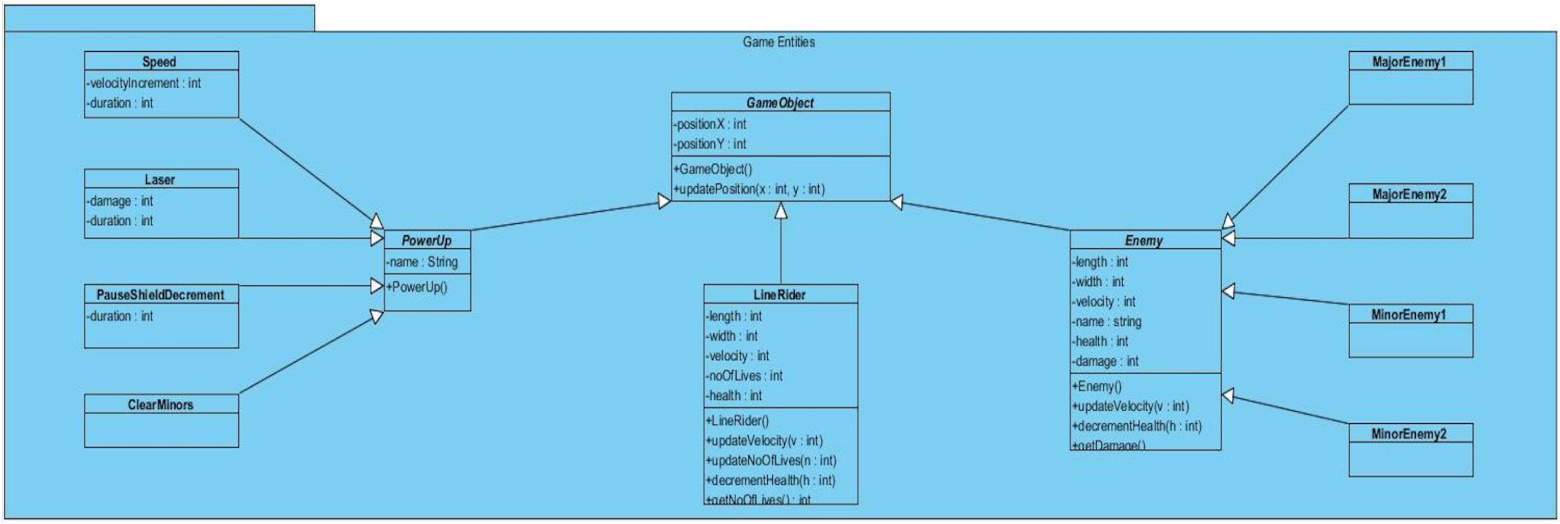


Figure 32. Game Entities Subsystem

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 Managament” and the bottom layer “Game Entitities”. 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 Managament”, 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.

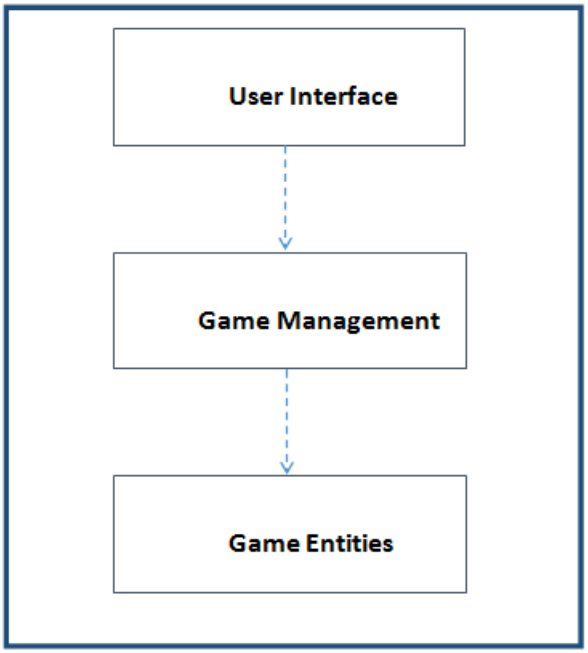


Figure 33. Hierarchical layers of system

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.

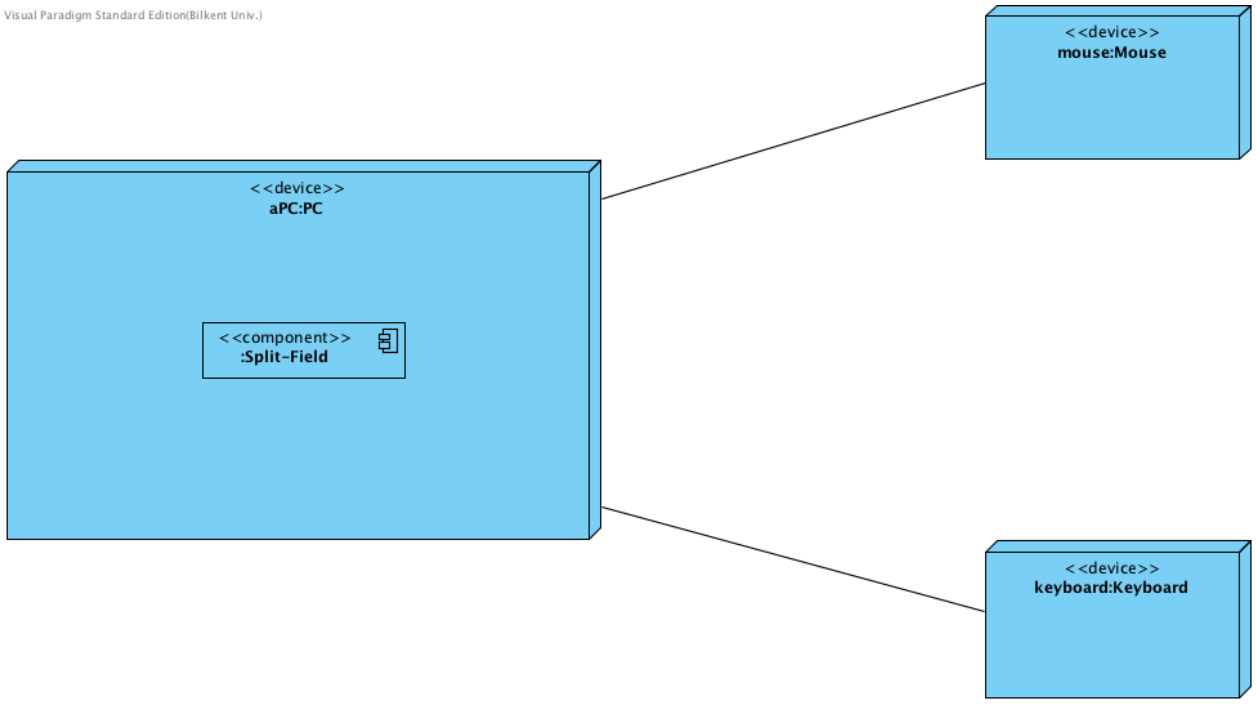


Figure 34. Deployment diagram of Split-Field

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.

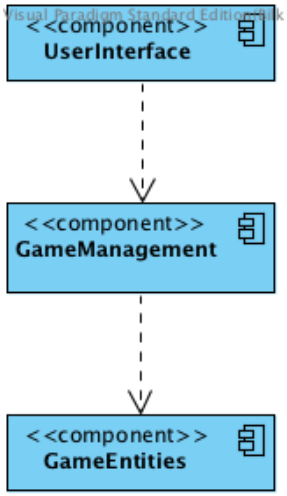


Figure 35. Component diagram of Split-Field component in the Deployment Diagram

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.

Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering Project

CS 319 Project: Split-Field

Final Report

Ali İlteriş TABAK, Efe TUNÇGENÇ, Gizem ÇAYLAK

Course Instructor: Hüseyin Özgür TAN

Progress

Dec 7, 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

6. Object Design

6.1. Pattern applications

In Split-Field game, we have applied three main design patterns to game's design:

- Facade Design Pattern
- Template Method Design Pattern
- Factory Method Pattern

Below we have explained how we applied these patterns to our system and the reason behind specific choice of these patterns to apply.

6.1.1. Facade Design Pattern

Facade pattern is a structural design pattern that provides a unified interface to a larger body of the code, set of objects in a subsystem. Facade pattern make these set of objects easier to use, understand and test, since the facade has convenient methods for common tasks by defining higher-level interface.

In our system design, we have used model-view-controller as an architectural pattern of our system. Among these classes, we needed communication since the connection between controller -view, and model- controller are not sufficient to reduce complexity. Thus, we have used Facade pattern to our system design to handle the couplings and reduce complexity among MVC classes as shown in Figure 36. In addition to MVC classes, we have provided a "Starter" class which is connected with view class("GameScreenManager"), model class("BackgroundManager"), and controller class("GameController") separately.

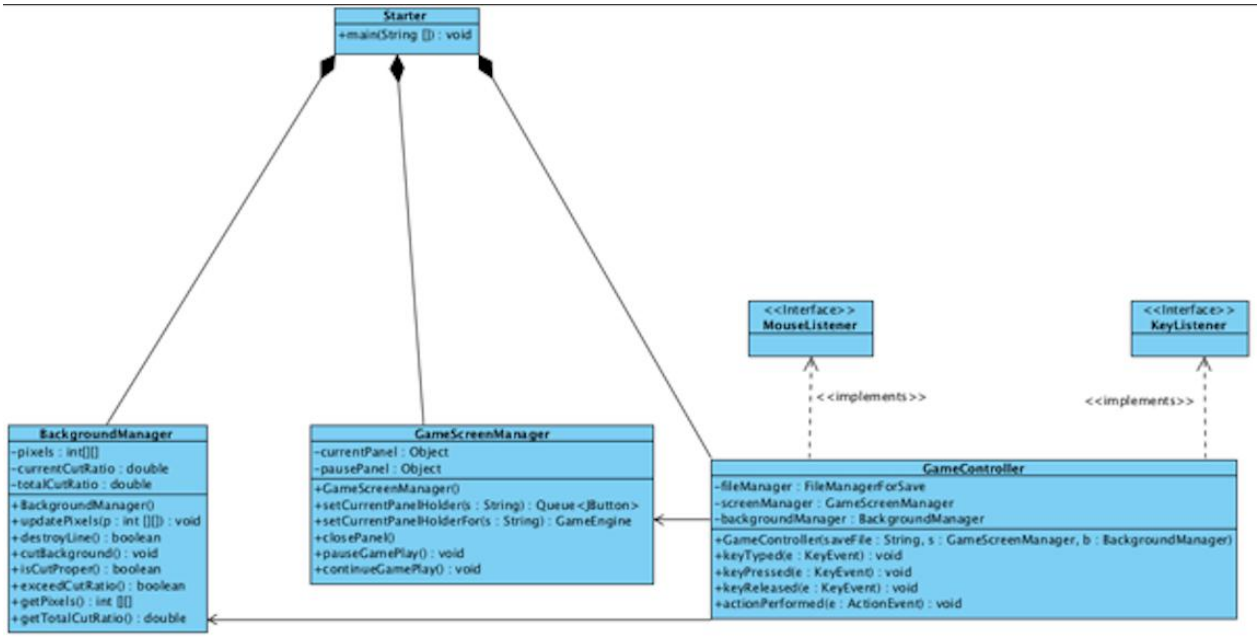


Figure 36 Facade design pattern for connection among model-view-controller architecture

6.1.2. Template Method Design Pattern

Template method design pattern is a behavioral design pattern that defines the skeleton of an algorithm in a method which is called template method. This pattern let one override certain steps of an algorithm without changing the structure of the algorithm. As shown in the Figure 37, we have applied this pattern on "File Management" subsystem within "Game Management" subsystem for three main reasons:

- It let subclasses implement behavior that can vary so "FileManagerForLoad", "FileManagerForSave" and "FileManagerForLevel" classes will be able to behave differently even if they have common ancestors.
- Duplication of code is avoided with this pattern since the general workflow structure is implemented once in abstract class's algorithm("*FileManager*"), and necessary variations are implemented in each of the subclasses of "*FileManager*".
- It provides control at points subclassing is allowed, as opposed to simple polymorphic override, in this pattern only the specific details of the workflow, in this case abstract "execute" function of "*FileManager*" abstract class, are allowed to change.

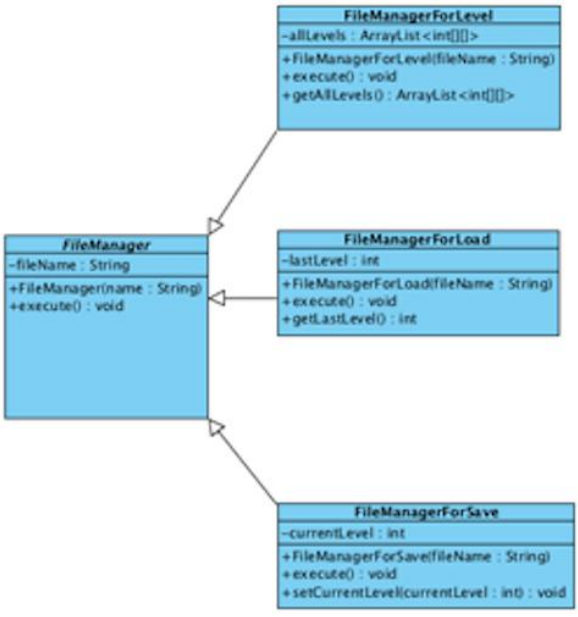


Figure 37 Template method pattern for the "File Management" subsystem.

6.1.3. Factory Method Pattern

Factory Method Pattern is a creational design pattern that uses factory methods to handle the creating objects problem without having to specify the class of the object that will be created. Rather than calling by constructor, this problem is handled by calling a factory method.

We have used factory method pattern on our “Game Entities” subsystem as shown in the Figure 38. As the first step, we have determined “GameObject” class as abstract class as a common super class. As the second step, “Enemy”, which consists of “MajorEnemy1”, “MajorEnemy2”, “MinorEnemy1”, “MinorEnemy2” classes, “PowerUp” ,which consists of “Speed”, “Laser”, “PauseShieldDecrement”, “ClearMinors” classes, and “LineRider” as classes inhering the same abstract class. As third step, we have created a “GameObjectFactory” class to generate object of concrete classes which are “MajorEnemy1”, “MajorEnemy2”, “MinorEnemy1”, “MinorEnemy2” , “Speed”, “Laser”, “PauseShieldDecrement”, “ClearMinors”, “LineRider” classes based on given information from “GameEngine” class. Finally, we can use “GameObjectFactory” class to get objects of concrete class by calling its function with string parameter indicating desired objects name(signature) within “GameEngine” class.

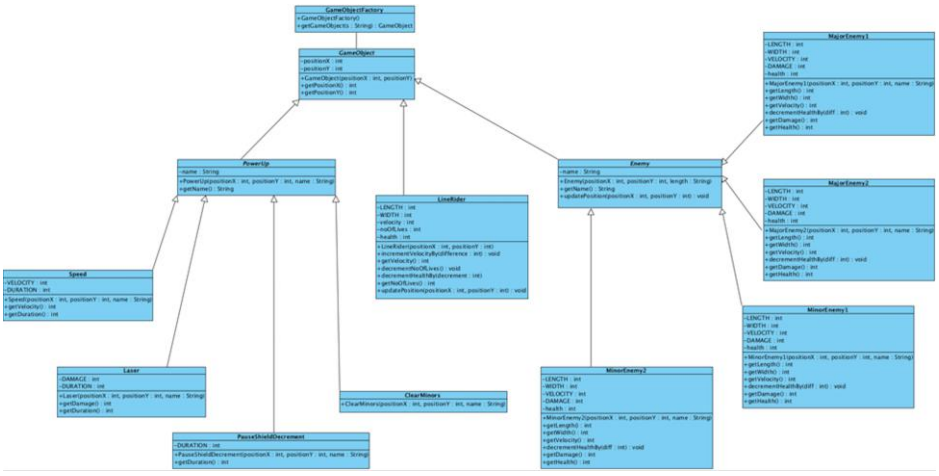


Figure 38 Factory method pattern for the “Game Entities” subsystem .

6.1.4. Pattern Applied Class Diagram

We have explained the design patterns, their logic and suitability above. We have applied these patterns to our system design and derived new class diagram as shown in the Figure 39.

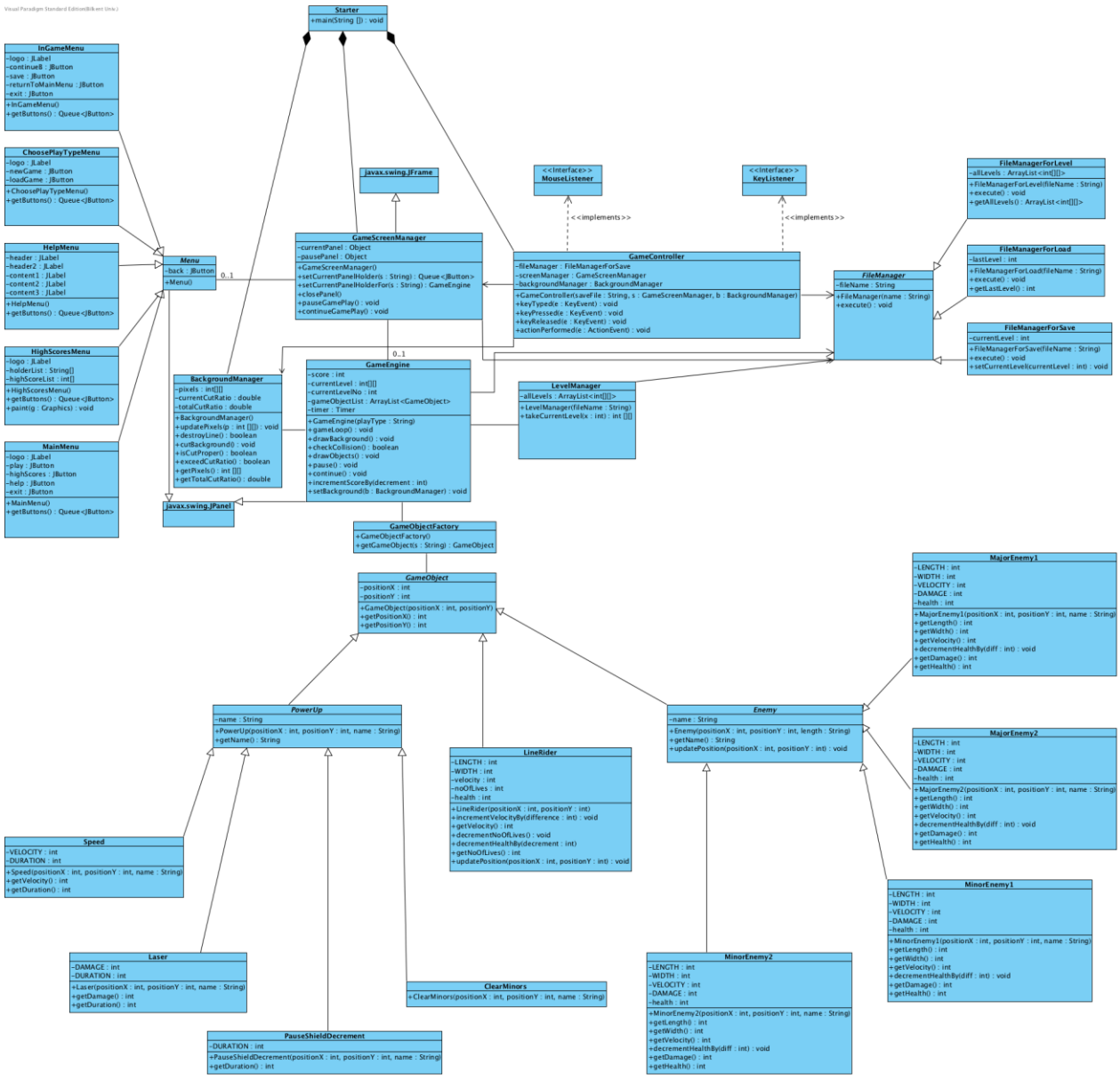


Figure 39 Patterns applied class diagram.

6.2. Class Interfaces

In this section, description of the classes, their public interfaces, which show type signature and visibility of methods that belong to the class, aim and usage of attributes, constructors are presented.

Starter Class

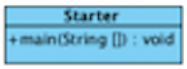


Figure 40 Starter class

Starter class can be considered as the initiator of the system. Because it hides the complexity of the system and provides a simplified interface of overall system, it can be considered as facade object that provides an entrance to the system.

Attributes:

-

Constructor:

-

Methods:

public static void main(String[] args): The main method is used for initiating the system. It creates a GameScreenManager object, a BackgroundManager object. These objects are given as parameters to constructor of GameController class.

GameController Class

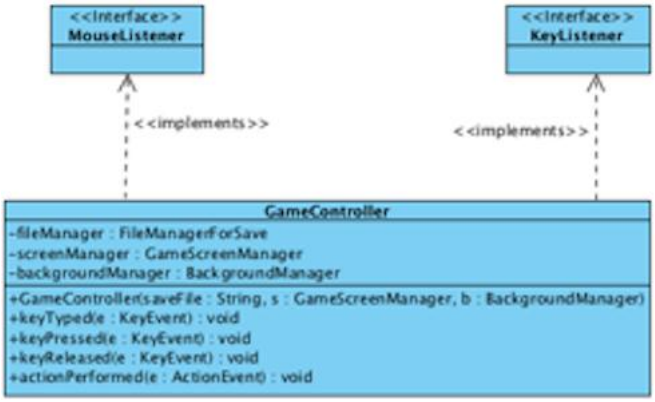


Figure 41 GameController class

GameController class can be considered as the controller of MVC architectural style. It implements KeyListener and ActionListener interfaces. The aim of this class is managing user inputs and directing the system according to these inputs.

Attributes:

private FileManagerForSave fileManager: The instance of FileManagerForSave class is used for holding the FileManagerForSave object created in GameController constructor. The object is used when a player clicks on "Save Game" button that exists in the in-game menu.

private GameScreenManager screenManager: The instance of GameScreenManager class is used for holding the GameScreenManager object which is taken as a parameter in constructor. It is used for sending the message for changing view.

private BackgroundManager backgroundManager: The instance of BackgroundManager class is used for holding the BackgroundManager object which is taken as a parameter in constructor. It can be considered as the model of the MVC architectural style. The state of model (BackgroundManager object) is changed according to the input of user during the gameplay.

Constructor:

public GameController(String saveFileName, GameScreenManager screenManager, BackgroundManager backgroundManager): Constructor of GameController class takes a string, that indicates the name of .txt file to save the last played level, to create a FileManagerForSave object, a GameScreenManager object and a BackgroundManager object.

Methods:

public void keyTyped(KeyEvent e): This function comes with the KeyListener interface but it is not used in the system.

public void keyPressed(KeyEvent e): This function is used for catching the keyboard inputs of the player during the gameplay. State of the model (BackgroundManager object) is determined according to the keyboard inputs.

public void keyReleased(KeyEvent e): This function comes with the KeyListener interface but it is not used in the system.

public void actionPerformed(ActionEvent e): This function is used for catching pressed buttons by the player in any menu. Clicking on a button triggers the action listener that calls this function. Navigation in the system is provided by actionPerformed function.

GameScreenManager Class

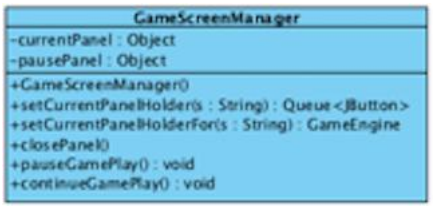


Figure 42 GameScreenManager class

GameScreenManager class extends JFrame class. It can be considered as holder, bridge between the controller (GameController) and the view (all menu classes and GameEngine class). GameScreenManager object manages the view objects according to the user inputs handled in the GameController object.

Attributes:

private Object currentPanel: It is used for holding the view object that is currently shown to the player. For example, when a player starts the system, currentPanel holds a MainMenu object which is the starting point of the navigational path.

private Object pausePanel: It is used for holding GameEngine object when the player pauses the game. That provides holding GameEngine object with its last played state while the system displays another view (in-game menu).

Constructors:

public GameScreenManager(): Constructor of GameScreenManager does not take any parameter. Adjustment of the JFrame is made in here.

Methods:

public Queue<JButton> setCurrentPanelHolder(String s): This function is called by GameController object when a player clicks on a button that navigates him to another menu view. It takes the menu name as a String parameter and changes the displayed view object according to the state of currentPanel. For example, pausePanel is only used when currentPanel holds a GameEngine object. It returns Queue<JButton> object, which is obtained by calling getButtons function of any menu class, actionPerformed method of GameController class. GameController object is added as ActionListener to JButton objects taken from setCurrentPanelHolder method of GameScreenManager.

public GameEngine setCurrentPanelHolderFor(String s): This function is called by GameController object when a player clicks on a button that navigates him to gameplay. It takes the play type as a String parameter and changes the currentPanel from ChoosePlayTypeMenu object to GameEngine object. It returns created GameEngine object to actionPerformed method of GameController class. GameController object is added as KeyListener to GameEngine object which extends JPanel class.

public void closePanel(): This function is called for removing any view object being held at currentPanel, when a player clicks on the "Exit" button.

public void pauseGameplay(): This function is called by GameController object when a player gives input to pause the game. The function calls the pause function of GameEngine object which is currently stored in the currentPanel attribute of GameScreenManager object.

public void continueGameplay(): This function is called by GameController object when a player gives input to continue the game. The function calls the continue function of GameEngine object which is currently stored in the currentPanel attribute of GameScreenManager object.

BackgroundManager Class

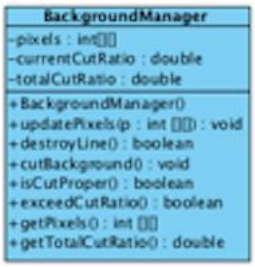


Figure 43 BackgroundManager class

BackgroundManager class can be considered as the model of MVC architectural style. State of the model (BackgroundManager object), which is demonstrated in the view, is determined by the user inputs handled by the controller (GameController object).

Attributes:

private int pixels[][]: This attribute is used for holding a two-dimensional integer array which is used for representing the game field that is required a player to cut. Initialization of the array is done in constructor of BackgroundManager class.

private double currentCutRatio: This attribute is used for holding a double that represents the ratio of currently cut background by the player. If cut is proper, currentCutRatio will be added to totalCutRatio.

private double totalCutRatio: This attribute is used for holding a double that represents the ratio of total cut background.

Constructors:

public BackgroundManager(): Constructor of BackgroundManager class does not take any parameter. Initialization of two-dimensional integer array is done. “totalCutRatio” and “currentCutRatio” attributes are set to zero.

Methods:

public void updatePixels(int p[][]): This function is used for updating the two-dimensional integer array that represents the game field. The function takes two-dimensional integer array as a parameter that is used for changing the state of model. State of model is represented in the array with different integers. For example, 1 represents the uncut area, 0 represents the cut area and 2 represents the line drawn by the player.

public boolean destroyLine(): If requirements of cutting an area are not satisfied by the player, the line, which is represented with 2's in the two-dimensional integer array, will be destroyed.

public void cutBackground(): If requirements of cutting an area are satisfied by the player, the line, which is represented with 2's in the two-dimensional integer array, will be added to cut area which is represented with 0's.

public boolean isCutProper(): Before calling cutBackground or destroyLine functions, isCutProper function is called to understand the properness of line drawn by the player. Decision about the state of the model is made according to the boolean returned by isCutProper function.

public boolean exceedCutRatio(): This function is called by GameEngine object for determining the end of the currentLevel. If totalCutRatio is greater than or equal to eighty, the function will return true that indicates the successful end of current level.

public int[][] getPixels(): The aim of this function is taking the two-dimensional integer array that represents the game field. The function is called by GameEngine object to draw the background.

public double getTotalCutRatio(): This function is called by GameEngine object for taking the totalCutRatio which is used for determining the flow of game and screening level score achieved by the player.

GameEngine Class

Figure 44 GameEngine class

GameEngine class extends JPanel class. It can be considered as view of MVC architectural style. GameEngine object is created and assigned to currentPanel attribute of GameScreenManager object when a player clicks on "New Game" or "Load Game" button. GameEngine class is in charge of gameplay that contains game logic, manipulation of game entities and creation of game field.

Attributes:

private int score: This attribute is used for holding the score of current player.

private int currentLevel[][]: This attribute is used for holding a two-dimensional integer array which is a numerical representation of current level created by GameEngine. Initialization of the array is done in constructor of GameEngine class via LevelManager object. CurrentLevel array is initialized through the gameplay as the player passes the levels.

private ArrayList<GameObject> gameObjectList: This attribute is used for holding an ArrayList object that stores GameObject objects acquired via getGameObject function of GameObjectFactory class.

private int currentLevelNo: This attribute is used for holding an integer that represents the current level played by the player. As player successfully finishes the levels, integer initially assigned in the constructor increments by one.

private LevelManager levelManager: The instance of LevelManager class is used for holding the LevelManager object created in GameEngine constructor. Level information like the kinds of enemies, power ups, their quantities and positions are provided by this object via a two-dimensional integer array (currentLevel[][]).

private FileManagerForLoad loadManager: The instance of FileManagerForLoad class is used for holding the FileManagerForLoad object created in GameEngine constructor. The object is used if the player wants to play where the last player stays. In this situation, FileManagerForLoad object reads the last played level from a .txt file. Returned integer is assigned to currentLevel attribute of GameEngine class and level information (two-dimensional integer array) is taken according to currentLevel.

private Timer timer: The instance of Timer class is used for holding a Timer object created in GameEngine constructor. Timer object is used for handling pause-continue states of the gameplay.

private GameObjectFactory factory: The instance of GameObjectFactory is used for holding a GameObjectFactory object that is created in the constructor of GameEngine class. The aim of GameObjectFactory object is acquiring the game entities with an easier and organized way.

Constructors:

public GameEngine(String playType): Constructor of GameEngine class take play type as a string parameter which indicates the choice of player. If a player clicks on “New Game” button in choose play type menu, actionPerformed function of GameController object calls the

setCurrentPanelFor function of ScreenManager object in that a new GameEngine object is declared with “new” argument, which corresponds to playType parameter, and assigned to currentPanel. In the light of acquired two-dimensional integer array, game entities are created using getGameObject function of GameObjectFactory class. In addition, adjustment of the JPanel is made in here.

Methods:

public void gameLoop(): This function contains a while statement. In each loop of while statement, game logic related functions of BackgroundManager object and GameObject objects are called. Additionally, gameLoop function calls required methods of GameEngine in order to draw the game field according to the model object (BackgroundManager object) and game entity objects (GameObject objects).

public void drawBackground(): This function is called by gameLoop method of GameEngine object in order to draw the model object (BackgroundManager object).

public boolean checkCollision(): This function is called by gameLoop method of GameEngine object to check the collision between LineRider object and any Enemy object. Collision check is done via comparing the position attributes of LineRider object and Enemy objects.

public void drawObjects(): Flow of events in the gameplay changes the states of game entities. Updated positions and states of game entities is used for redrawing them to JPanel.

public void pause(): This method is called by pauseGameplay method of GameScreenManager object. That changes the state of Timer object, which is held in timer attribute of GameEngine class, to change the behavior of while statement in the gameLoop function.

public void continue(): This method is called by continueGameplay method of GameScreenManager object. That changes the state of Timer object, which is held in timer

attribute of GameEngine class, to change the behavior of while statement in the gameLoop function.

public void incrementScoreBy(int decrement): This method is called by gameLoop method of GameEngine class. That helps organizing the game logic by handling with the score of player.

public void setBackgroundManager(BackgroundManager backgroundManager): This method is called by GameController object after GameScreenManager object declares a new GameEngine class. The aim of this method can be considered as connecting the model (BackgroundManager object) and view (GameEngine object) to each others. Required adjustment on BackgroundManager object made after the call of this function.

LevelManager Class

Figure 45 LevelManager class

LevelManager class is an assistant class of GameEngine class. It maintains the information of all levels and provides information of a specific level when GameEngine requests. LevelManager uses a FileManagerForLevel object to obtain the two-dimensional integer array that contains the current level information like kind and quantity of enemies, power ups.

Attributes:

private ArrayList<int[][]> allLevels: This attribute is used for holding an ArrayList object that stores two-dimensional integer arrays acquired via getLevels method of FilemanagerForLevel.

private FileManagerForLevel fileManager: The instance of FileManagerForLevel class is used for holding the FileManagerLevel object created in LevelManager constructor. The object is used for taking an ArrayList<int[][]> object from a txt file.

Constructors:

public LevelManager(String fileName): Constructor of LevelManager takes file name of txt file, that contains the information of all levels, as a string parameter. FileManagerForLevel object is declared using the fileName parameter and assigned to fileManager attribute of LevelManager object. Additionally, it calls firstly execute method then loadLevels method of FileManagerForLevel object to obtain the ArrayList<int[][]> object.

Methods:

public int[][] takeCurrentLevel(int levelNo): This function is called by GameEngine constructor before obtaining the GameObject objects from the GameObjectFactory in order to acquire the information of current level with respect to levelNo parameter.

FileManager Abstract Class

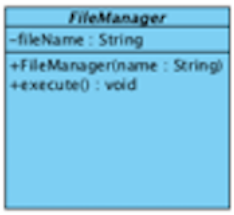


Figure 46 FileManager abstract class

As indicated in the title, FileManager is an abstract class. The aim of this class is being a template (Template Design Pattern) for its child classes which are FileManagerForLevel, FileManagerForSave and FileManagerForLoad.

Attributes:

private String fileName: This attribute is used for holding file name as a string. Content of this attribute is declared in constructor of FileManager abstract class.

Constructors:

public FileManager(String fileName): Constructor of FileManager abstract class takes file name as string parameter.

Methods:

public abstract void execute(): Because execute is an abstract method, its implementation is made in its child classes which are FileManagerForLevel, FileManagerForSave and FileManagerForLoad. In each class execute function is implemented according to the requirements of the aim of child classes. For example, execute function of FileManagerForLoad is implemented in a way that it can read from a txt file. However, execute function of FileManagerForSave is implemented in a way that it can write to a txt file.

FileManagerForLevel Class

Figure 47 FileManagerForLevel class

FileManagerForLevel class extends the FileManager abstract class and overrides constructor and execute method. It provides connection between LevelManager object, that holds information of all levels through the execution of system, and file, that contain numerical representation of all levels. FileManagerForLevel object is used by LevelManager object to take information of all levels as an ArrayList<int[][]> object.

Attributes:

private ArrayList<int[][]> allLevels: This attribute is used for holding an ArrayList object that stores a two-dimensional integer arrays. ArrayList<int[][]> object is assigned in "execute" method of FileManagerForLevel class.

Constructors:

public FileManagerForLevel(String fileName): Constructor of FileManagerForLevel class takes file name of the txt file in that numerical representation of all levels information is stored. Additionally, "execute" method of FileManagerForLevel class is called in order to prepare ArrayList<int[][]> object to any call of getAllLevels method.

Methods:

public void execute(): This method is called by constructor of FileManagerForLevel class. The method reads numerical representation of information of all levels from the file whose name is given as a parameter to constructor and places them in an ArrayList object of two-dimensional integer arrays.

public ArrayList<int[][]> getAllLevels(): This method is called by LevelManager object of GameEngine class to obtain the information of all levels. It returns the ArrayList<int[][]> object held in allLevels attribute of FileManagerForLevel class to LevelManager.

FileManagerForLoad Class

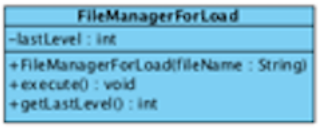


Figure 48 FileManagerForLoad class

FileManagerForLoad class extends the FileManager abstract class and overrides constructor and execute method. It provides connection between GameEngine object, that requires the information of last played level to create game field, and file, that contains numerical representation of last played level.

Attributes:

private int lastLevel: This attribute is used for holding numerical representation of last played level which is assigned in execute method of FileManagerForLoad class.

Constructors:

public FileManagerForLoad(String fileName): Constructor of FileManagerForLoad class takes file name of the txt file in that numerical representation of last played level is stored. Additionally, "execute" method of FileManagerForLoad class is called in order to prepare integer that represents the information of last played level to any call of getLastLevel method.

Methods:

public void execute(): This method is called by constructor of FileManagerForLoad class. The method reads numerical representation of last played level from the file whose name is given as a parameter to constructor and assigned it to lastLevel attribute of FileManagerForLoad class.

public int getLastLevel(): This method is called by GameEngine object to obtain the numerical representation of last played level. Basically, it returns the value of lastLevel attribute of FileManagerForLoad class.

FileManagerForSave Class

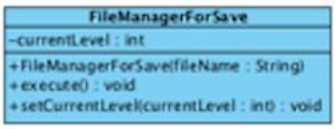


Figure 49 FileManagerForSave class

FileManagerForSave class extends the FileManager abstract class and overrides constructor and execute method. It provides connection between GameController object, that handles save request which is triggered by player via clicking on "Save Game" button at in-game menu, and file, that is used to store the numerical representation of currently played level.

Attributes:

private int currentLevel: This attribute is used for holding numerical representation of currently played level which is assigned in setCurrentlevel method of FileManagerForSave class.

Constructors:

public FileManagerForSave(String fileName): Constructor of FileManagerForSave class takes file name of the txt file in that numerical representation of currently played level will be stored. String parameter, fileName, is same with parameter given to FileManagerForLoad constructor which means system reads the last played level and writes currently played level to the same file.

Methods:

public void execute(): This method is called by setCurrentLevel method of FileManagerForSave class. The method writes numerical representation of currently played level to the file whose name is given as a parameter to constructor. The method can overwrite the number stored in the txt file if the game is previously played and its current level is saved.

public void setCurrentLevel(int currentLevel): This method is called by GameController object for transmitting numerical representation of currently played level. Integer parameter, currentLevel is assigned to currentLevel attribute of FileManagerForSave class. In addition, execute method of FileManagerForSave is called in here to trigger the writing process.

Menu Abstract Class

Figure 50 Menu abstract class

Menu abstract class extends JPanel class. It can be considered as view of MVC architectural style. The aim of this abstract class is being template to its child classes which are MainMenu class, InGameMenu class, HighScoresMenu class, HelpMenu class and ChoosePlayTypeMenu class.

Attributes:

private JButton back: Because "Back" button is used in many child classes of Menu abstract class, the instance of JButton class is used for holding the JButton object created in Menu constructor.

Constructors:

public Menu(): Constructor of Menu abstract class does not take any parameter. Adjustment of the JFrame is made in here.

Methods:

-

MainMenu Class

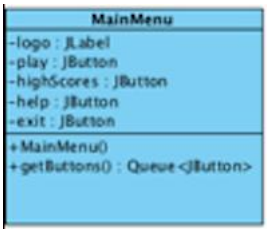


Figure 51 MainMenu class

MainMenu class extends Menu abstract class. It can be considered as view of MVC architectural style. This class is the first menu seen by the player when he runs the system. It contains buttons to navigate through system.

Attributes:

private JLabel logo: The instance of JLabel class is used for holding the JLabel object created in MainMenu constructor. JLabel object contains the name of the game to display “Split-Field” in the screen.

private JButton play: The instance of JButton class is used for holding the JButton object created in MainMenu constructor. Adjustment of JButton object is made in constructor of MainMenu class. Clicking on this button navigates player to choose play type menu. Navigation is managed by GameScreenManager object according to the user inputs handled by GameController object.

private JButton highScores: The instance of JButton class is used for holding the JButton object created in MainMenu constructor. Adjustment of JButton object is made in constructor of MainMenu class. Clicking on this button navigates player to high score menu. Navigation is managed by GameScreenManager object according to the user inputs handled by GameController object.

private JButton help: The instance of JButton class is used for holding the JButton object created in MainMenu constructor. Adjustment of JButton object is made in constructor of MainMenu class. Clicking on this button navigates player to help menu. Navigation is managed by GameScreenManager object according to the user inputs handled by GameController object.

private JButton exit: The instance of JButton class is used for holding the JButton object created in MainMenu constructor. Adjustment of JButton object is made in constructor of MainMenu class. Clicking on this button terminates the system.

Constructors:

public MainMenu(): Constructor of MainMenu does not take any parameter. Adjustment of the JPanel and adjustment of components of JPanel, which are listed as attributes of this class, is made in here. However, adding ActionListener to JButton is not done. That is handled in the GameController.

Methods:

public Queue<JButton> getButtons(): This method is called from setCurrentPanelHolder method of GameScreenManager class. Buttons of MainMenu class are added to a Queue<JButton> object. setCurrentPanelHolder also returns the Queue<JButton> object to actionPerformed method of GameController class. In actionPerformed method, GameController object is added as ActionListener to JButton objects that are dequeued from Queue<JButton> object.

InGameMenu Class

Figure 52 InGameMenu class

MainMenu class extends Menu abstract class. It can be considered as view of MVC architectural style. It contains buttons to navigate through system and label to show which menu the player sees. It is showed when the player pauses the gameplay.

Attributes:

private JLabel logo: The instance of JLabel class is used for holding the JLabel object created in InGameMenu constructor. JLabel object contains the text of "In Game Menu" to demonstrate in the screen.

private JButton continueB: The instance of JButton class is used for holding the JButton object created in InGameMenu constructor. Adjustment of JButton object is made in constructor of InGameMenu class. Clicking on this button navigates player to gameplay. Navigation is managed by GameScreenManager object according to the user inputs handled by GameController object.

private JButton save: The instance of JButton class is used for holding the JButton object created in InGameMenu constructor. Adjustment of JButton object is made in constructor of InGameMenu class. Clicking on this button triggers the saving process of currently played level.

private JButton returnToMainMenu: The instance of JButton class is used for holding the JButton object created in InGameMenu constructor. Adjustment of JButton object is made in constructor of InGameMenu class. Clicking on this button navigates player to main menu.

private JButton exit: The instance of JButton class is used for holding the JButton object created in InGameMenu constructor. Adjustment of JButton object is made in constructor of InGameMenu class. Clicking on this button terminates the system.

Constructors:

public InGameMenu(): Constructor of InGameMenu does not take any parameter. Adjustment of the JPanel and adjustment of components of JPanel, which are listed as attributes of this class, is made in here. However, adding ActionListener to JButton is not done. That is handled in the GameController.

Methods:

public Queue<JButton> getButtons(): This method is called from setCurrentPanelHolder method of GameScreenManager class. Buttons of InGameMenu class are added to a Queue<JButton> object. setCurrentPanelHolder also returns the Queue<JButton> object to actionPerformed method of GameController class. In actionPerformed method, GameController object is added as ActionListener to JButton objects that are dequeued from Queue<JButton> object.

HighScoresMenu Class

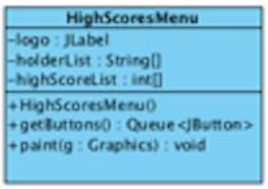


Figure 53 HighScoresMenu class

HighScoresMenu class extends Menu abstract class. It can be considered as view of MVC architectural style. It contains buttons to navigate through system and labels to show highscores and their holders. It is showed when the player clicks on "High Scores" button in main menu.

Attributes:

private JLabel logo: The instance of JLabel class is used for holding the JLabel object created in HighScoreMenu constructor. JLabel object contains the text of "High Scores" to demonstrate in the screen.

private String holderList[]: This attribute is used for holding String objects which indicate the names of owner of high score. Assignment to this attribute is done in constructor of HighScoresMenu contructor.

private int highScoresList[]: This attribute is used for holding numerical representations of high scores. Assignment to this attribute is done in constructor of HighScoresMenu contructor.

Constructors:

public HighScoreMenu(): Constructor of HighScoreMenu does not take any parameter. Adjustment of the JPanel and adjustment of components of JPanel, which are listed as attributes of this class, is made in here. However, adding ActionListener to JButton is not done. That is handled in the GameController.

Methods:

public void paint(Graphics g): This method is called from constructor of HighScoresMenu class. That provides drawing the high scores and their holders to panel.

public Queue<JButton> getButtons(): This method is called from setCurrentPanelHolder method of GameScreenManager class. Buttons of HighScoresMenu class are added to a Queue<JButton> object. setCurrentPanelHolder also returns the Queue<JButton> object to actionPerformed method of GameController class. In actionPerformed method, GameController object is added as ActionListener to JButton objects that are dequeued from Queue<JButton> object.

HelpMenu Class

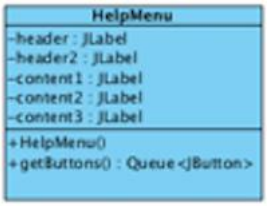


Figure 54 HelpMenu class

HelpMenu class extends Menu abstract class. It can be considered as view of MVC architectural style. It contains buttons to navigate through system and labels to show actions, that can be made by player, and their buttons. It is showed when the player clicks on “Help” button in main menu.

Attributes:

private JLabel header: The instance of JLabel class is used for holding the JLabel object created in HelpMenu constructor. JLabel object contains the text of “Help” to demonstrate in the screen.

private JLabel header2: The instance of JLabel class is used for holding the JLabel object created in HelpMenu constructor. JLabel object contains the text of “How to Play” to demonstrate in the screen.

private JLabel content1: The instance of JLabel class is used for holding the JLabel object created in HelpMenu constructor. JLabel object contains the text of “move : key arrows” to demonstrate in the screen.

private JLabel content2: The instance of JLabel class is used for holding the JLabel object created in HelpMenu constructor. JLabel object contains the text of “start cut: space” to demonstrate in the screen.

private JLabel content3: The instance of JLabel class is used for holding the JLabel object created in HelpMenu constructor. JLabel object contains the text of “laser cut: z” to demonstrate in the screen.

Constructors:

public HelpMenu(): Constructor of HelpMenu does not take any parameter. Adjustment of the JPanel and adjustment of components of JPanel, which are listed as attributes of this class, is made in here. However, adding ActionListener to JButton is not done. That is handled in the GameController.

Methods:

public Queue<JButton> getButtons(): This method is called from setCurrentPanelHolder method of GameScreenManager class. Buttons of HelpMenu class are added to a Queue<JButton> object. setCurrentPanelHolder also returns the Queue<JButton> object to actionPerformed method of GameController class. In actionPerformed method, GameController

object is added as ActionListener to JButton objects that are dequeued from Queue<JButton> object.

ChoosePlayTypeMenu Class

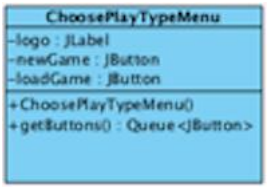


Figure 55 ChoosePlayTypeMenu class

ChoosePlayTypeMenu class extends Menu abstract class. It can be considered as view of MVC architectural style. It contains buttons to navigate through system and label to show which menu the player sees. It is showed when the player clicks on "Play Game" button in main menu.

Attributes:

private JLabel logo: The instance of JLabel class is used for holding the JLabel object created in ChoosePlayTypeMenu constructor. JLabel object contains the text of "Play Game" to demonstrate in the screen.

private JButton newGame: The instance of JButton class is used for holding the JButton object created in ChoosePlayTypeMenu constructor. Adjustment of JButton object is made in constructor of ChoosePlayTypeMenu class. Clicking on this button navigates player to gameplay that starts from the first level.

private JButton loadGame: The instance of JButton class is used for holding the JButton object created in ChoosePlayTypeMenu constructor. Adjustment of JButton object is made in constructor of ChoosePlayTypeMenu class. Clicking on this button navigates player to gameplay that starts from the last level played.

Constructors:

public ChoosePlayTypeMenu(): Constructor of ChoosePlayTypeMenu does not take any parameter. Adjustment of the JPanel and adjustment of components of JPanel, which are listed as attributes of this class, is made in here. However, adding ActionListener to JButton is not done. That is handled in the GameController.

Methods:

public Queue<JButton> getButtons(): This method is called from setCurrentPanelHolder method of GameScreenManager class. Buttons of ChoosePlayTypeMenu class are added to a Queue<JButton> object. setCurrentPanelHolder also returns the Queue<JButton> object to actionPerformed method of GameController class. In actionPerformed method, GameController object is added as ActionListener to JButton objects that are dequeued from Queue<JButton> object.

GameObjectFactory Class

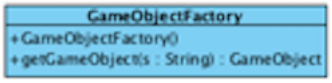


Figure 56 GameObjectFactory class

This class forms the base of Factory pattern and will be used to create Game Objects by specifying the name of the object that will be created as a string. “GameEngine” class will have access to this class’s content to create GameObjects.

Attributes:

-

Constructors:

-

Methods:

public GameObject getGameObject(String gameObjectKind): This method will be used to get a GameObject. “GameEngine” will call this method by entering the name of the object that is desired as string parameter and this method will return the desired GameObject according to string parameter.

GameObject Class

Figure 57 GameObject class

This class is an abstract class that acts as a prototype for PowerUp, Enemy, LineRider classes. This class cannot be instantiated since it requires special properties provided by its subclasses to have role in game field.

Attributes:

protected int positionX: integer variable that holds the X coordinate of the GameObject in the game field.

protected int positionY: integer variable that holds the Y coordinate of the GameObject in the game field.

Constructors:

public GameObject(int positionX, int positionY): initialize the X and Y coordinates of the GameObject according to given coordinate parameters.

Methods:

public int getPositionX(): returns the X coordinate of the "GameObject" object in the game field.

public int getPositionY(): returns the Y coordinate of the "GameObject" object in the game field.

PowerUp Class

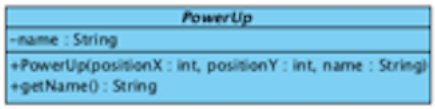


Figure 58 PowerUp class

This class is an abstract class that extends "GameObject" class and acts as a prototype for Speed, Laser, PauseShieldDecrement, and ClearMinor classes. This class cannot be instantiated since it requires special properties provided by its subclasses to have role, which is enhancing the LineRider's properties, in the game field. There could be zero or more "PowerUp" objects in the game field.

Attributes:

private String name: string variable that holds the name value of the PowerUp instances that will be used by "GameObjectFactory" to create these objects.

Constructors:

public PowerUp(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class "GameObject" according to given coordinate parameters, and name of the "PowerUp" instances that will be used by "GameObjectFactory" to create these objects.

Methods:

public String getName(): returns the name of the "PowerUp" object.

Speed Class

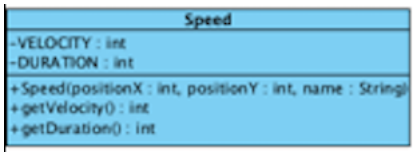


Figure 59 Speed class

"Speed" class extends "PowerUp" class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the "GameObject" class. The "LineRider" can use "Speed" object to enhance its velocity property by adding the specified velocity attribute of "Speed" object to its velocity attribute.

Attributes:

private static final int VELOCITY: a constant integer variable that holds the velocity amount of the "Speed" object that will enhance the velocity of the "LineRider" initialized to 3.

private static final int DURATION: a constant integer variable that holds the amount of time in seconds that the "LineRider" can use this power up initialized to 5.

Constructors:

public Speed(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class "PowerUp" according to given coordinate parameters, and name of the "Speed" instances that will be used by "GameObjectFactory" to create these objects.

Methods:

public int getVelocity(): returns the integer value of "VELOCITY" attribute that holds the velocity amount of the "Speed" object.

public int getDuration():returns the integer value of "DURATION" attribute that holds the amount of time in seconds that the "LineRider" can use this power up.

Laser Class

Figure 60 Laser class

"Laser" class extends "PowerUp" class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the "GameObject" class. The "LineRider" can use "Laser" object to gain laser cut power which will activate its damage attribute by assigning "DAMAGE" value of "Laser" class to its "damage" variable to damage enemies.

Attributes:

private static final int DAMAGE: a constant integer variable that holds the damage amount which "LineRider" object will gain with this "PowerUp" object to damage enemies initialized to 15.

private static final int DURATION: a constant integer variable that holds the amount of time in seconds that the "LineRider" can use this power up initialized to 5.

Constructors:

public Laser(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class "PowerUp" according to given coordinate parameters, and name of the "Laser" instances that will be used by "GameObjectFactory" to create these objects.

Methods:

public int getDamage():returns the integer value of "DAMAGE" attribute that holds the damage amount which "LineRider" object will gain with this "PowerUp" object to damage enemies

public int getDuration():returns the integer value of "DURATION" attribute that holds the amount of time in seconds that the "LineRider" can use this power up.

PauseShieldDecrement Class

Figure 61 PauseShieldDecrement class

"PauseShieldDecrement" class extends "PowerUp" class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the "GameObject" class. The "LineRider" can use "PauseShieldDecrement" object to reset to zero enemies' velocity in a way stop time for enemy objects while "LineRider" can move and cut area in game field.

Attributes:

private static final int DURATION: a constant integer variable that holds the amount of time in seconds that the "LineRider" can use this power up initialized to 10.

Constructors:

public PauseShieldDecrement(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class "PowerUp" according to given coordinate parameters, and name of the "PauseShieldDecrement" instances that will be used by "GameObjectFactory" to create these objects.

Methods:

public int getDuration():returns the integer value of "DURATION" attribute that holds the amount of time in seconds that the "LineRider" can use this power up.

ClearMinors Class

Figure 62 ClearMinor class

"ClearMinors" class extends "PowerUp" class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the "GameObject" class. The "LineRider" can use "ClearMinors" object to clear the minor enemies, which can be MinorEnemy1 or MinorEnemy2, in the game field.

Attributes:

-

Constructors:

public ClearMinors(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class "PowerUp", and name of the "ClearMinor" instances that will be used by "GameObjectFactory" to create these objects.

Methods:

-

LineRider Class

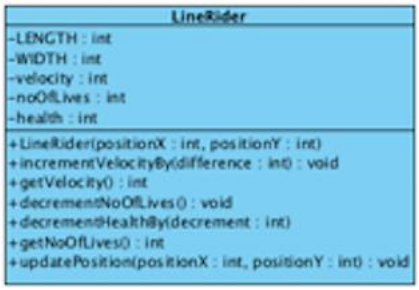


Figure 63 LineRider class

"LineRider" class extends "GameObject" class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the "GameObject" class. There could be only 1 "LineRider" object in the game field. This class represents the game character of the Split-Field game and is controlled by the user via special keyboard keys.

Attributes:

private static final int LENGTH: a constant integer variable that holds the length of the "LineRider" initialized to 5.

private static final int WIDTH: a constant integer variable that holds the width of the "LineRider" initialized to 5.

private int velocity: a integer variable that holds the velocity of the "LineRider".

private int noOfLives: a integer variable that holds the number of lives of the "LineRider" which could be maximum 3.

private int health: a integer variable that holds the health value of the "LineRider" which could be maximum 100.

Constructors:

public LineRider(int positionX, int positionY): initialize the X and Y coordinates that is inherited from its super class "GameObject" according to given coordinate parameters and set initial velocity, noOfLives and health variables of "LineRider" object respectively to 4, 3, 100.

Methods:

public void incrementVelocityBy(int difference): this method increase the velocity of the "LineRider" object by adding given parameter "difference" to it. This method is invoked by "Speed" class which is a power up that enhance the velocity of the "LineRider" object.

public int getVelocity():returns the integer value of "velocity" attribute of "LineRider" object.

public void decrementNoOfLives(): this method decrease the number of lives of "LineRider" object by 1, when the health attribute is less than 0.

public void decrementHealthBy(int decrement): this method decrease the "health" of "LineRider" object by subtracting "decrement" parameter which is determined by the damage amount of enemy which collides with "LineRider" object in game field.

public int getNoOfLives(): this method returns the integer value of "noOfLives" attribute.

public void updatePosition(int positionX, int positionY): this method updates the X and Y coordinates of "LineRider" object which stems from player keyboard inputs to move the "LineRider" object.

Enemy Class

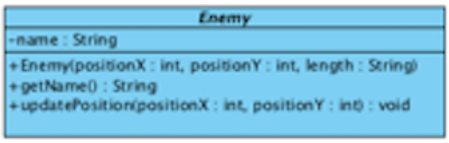


Figure 64 Enemy class

This class is an abstract class that extends "GameObject" class and acts as a prototype for "MajorEnemy1", "MajorEnemy2", "MinorEnemy1", and "MinorEnemy2" classes. This class cannot be instantiated since it requires special properties provided by its subclasses to have role, which is trying to prevent "LineRider" object cutting the field by damaging the "LineRider" object or blocking the movement of it, in the game field. There could be one or more "Enemy" objects in the game field. Major enemies are superior than minor enemies in terms of velocity, length, width, damage properties. In addition to superiority difference between major and minor enemies, minor enemies can be cleared via "ClearMinors" power up.

Attributes:

protected String name: string variable that holds the name value of the "Enemy" instances that will be used by "GameObjectFactory" to create these objects.

Constructors:

public Enemy(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class "GameObject" according to given coordinate parameters, and name of the "Enemy" instances that will be used by "GameObjectFactory" to create these objects.

Methods:

public String getName():returns the name of the “Enemy” object.

public void updatePosition(int positionX, int positionY): this method updates the X and Y coordinates of the “Enemy” object which is done by an algorithm by the system in sync with “LineRider”’s movements that is controlled by the player.

MajorEnemy1 Class

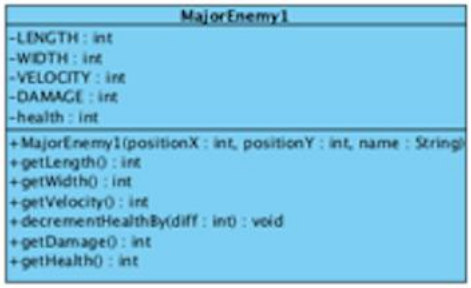


Figure 65 MajorEnemy1 class

“MajorEnemy1” class extends “Enemy” class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the “GameObject” class. “MajorEnemy1” has the same effect on “LineRider” class in terms of velocity and damage comparing to “MajorEnemy2” but they differ in length and width properties.

Attributes:

private static final int LENGTH: a constant integer variable that holds the length of the “LineRider” initialized to 20.

private static final int WIDTH: a constant integer variable that holds the width of the “LineRider” initialized to 10.

private static final int VELOCITY: a constant integer variable that holds the velocity of the "Enemy" initialized to 5..

private static final int DAMAGE: a constant integer variable that holds the damage amount which "Enemy" object will cause decrease in the health of the "LineRider" object initialized to 5.

private int health: a integer variable that holds the health value of the "Enemy" which could be maximum 100.

Constructors:

public MajorEnemy1(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class "Enemy" according to given coordinate parameters, and name of the "MajorEnemy1" instances that will be used by "GameObjectFactory" to create these objects.

Methods:

public int getLength(): returns the value of length attribute of "MajorEnemy1" object.

public int getWidth(): returns the value of width attribute of "MajorEnemy1" object.

public int getVelocity(): returns the value of velocity attribute of "MajorEnemy1" object.

public void decrementHealthBy(int diff):): this method decrease the "health" of "MajorEnemy1" object by subtracting "diff" parameter which is determined by the damage amount of "LinerRider" which damage the enemy with lasercut powerup.

public int getDamage(): returns the value of damage attribute of "MajorEnemy1" object.

public int getHealth():returns the value of health attribute of "MajorEnemy1" object.

MajorEnemy2 Class

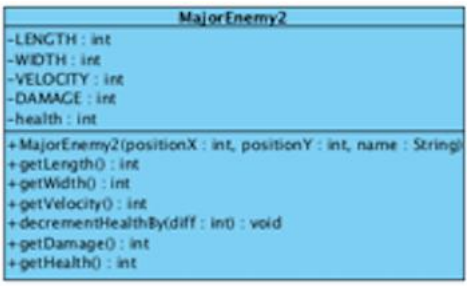


Figure 66 MajorEnemy2

"MajorEnemy2" class extends "Enemy" class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the "GameObject" class. "MajorEnemy2" has the same effect on "LineRider" class in terms of velocity and damage comparing to "MajorEnemy1" but they differ in length and width properties.

Attributes:

private static final int LENGTH: a constant integer variable that holds the length of the "LineRider" initialized to 15.

private static final int WIDTH: a constant integer variable that holds the width of the "LineRider" initialized to 5.

private static final int VELOCITY: a constant integer variable that holds the velocity of the "Enemy" initialized to 5..

private static final int DAMAGE: a constant integer variable that holds the damage amount which "Enemy" object will cause decrease in the health of the "LineRider" object initialized to 5.

private int health: a integer variable that holds the health value of the "Enemy" which could be maximum 100.

Constructors:

public MajorEnemy2(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class “Enemy” according to given coordinate parameters, and name of the “MajorEnemy2” instances that will be used by “GameObjectFactory” to create these objects.

Methods:

public int getLength():returns the value of length attribute of “MajorEnemy2” object.

public int getWidth():returns the value of width attribute of “MajorEnemy2” object.

public int getVelocity():returns the value of velocity attribute of “MajorEnemy2” object.

public void decrementHealthBy(int diff): this method decrease the “health” of “MajorEnemy2” object by subtracting “diff” parameter which is determined by the damage amount of “LinerRider” which damage the enemy with lasercut powerup.

public int getDamage():returns the value of damage attribute of “MajorEnemy2” object.

public int getHealth():returns the value of health attribute of “MajorEnemy2” object.

MinorEnemy1 Class

Figure 67 MinorEnemy1

"MinorEnemy1" class extends "Enemy" class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the "GameObject" class. "MinorEnemy1" has the same effect on "LineRider" class in terms of velocity and damage comparing to "MinorEnemy2" but they differ in length and width properties.

Attributes:

private static final int LENGTH: a constant integer variable that holds the length of the "LineRider" initialized to 5.

private static final int WIDTH: a constant integer variable that holds the width of the "LineRider" initialized to 5.

private static final int VELOCITY: a constant integer variable that holds the velocity of the "Enemy" initialized to 2.

private static final int DAMAGE: a constant integer variable that holds the damage amount which "Enemy" object will cause decrease in the health of the "LineRider" object initialized to 2.

private int health: a integer variable that holds the health value of the "Enemy" which could be maximum 100.

Constructors:

public MinorEnemy1(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class “Enemy” according to given coordinate parameters, and name of the “MinorEnemy1” instances that will be used by “GameObjectFactory” to create these objects.

Methods:

public int getLength():returns the value of length attribute of “MinorEnemy1” object.

public int getWidth():returns the value of width attribute of “MinorEnemy1” object.

public int getVelocity():returns the value of velocity attribute of “MinorEnemy1” object.

public void decrementHealthBy(int diff): this method decrease the “health” of “MinorEnemy1” object by subtracting “diff” parameter which is determined by the damage amount of “LinerRider” which damage the enemy with “LaserCut” powerup or reset to zero by “ClearMinor” powerup.

public int getDamage():returns the value of damage attribute of “MinorEnemy1” object.

public int getHealth():returns the value of health attribute of “MinorEnemy1” object.

MinorEnemy2 Class

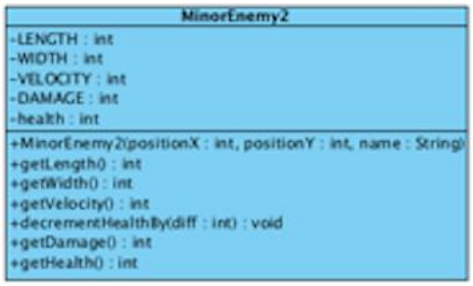


Figure 68 MinorEnemy2

"MinorEnemy2" class extends "Enemy" class. This class can be instantiated to have role in game field since it is one of the specialized class in inheritance tree of the "GameObject" class. "MinorEnemy2" has the same effect on "LineRider" class in terms of velocity and damage comparing to "MinorEnemy1" but they differ in length and width properties.

Attributes:

private static final int LENGTH: a constant integer variable that holds the length of the "LineRider" initialized to 3.

private static final int WIDTH: a constant integer variable that holds the width of the "LineRider" initialized to 3.

private static final int VELOCITY: a constant integer variable that holds the velocity of the "Enemy" initialized to 2.

private static final int DAMAGE: a constant integer variable that holds the damage amount which "Enemy" object will cause decrease in the health of the "LineRider" object initialized to 2.

private int health: a integer variable that holds the health value of the "Enemy" which could be maximum 100.

Constructors:

public MinorEnemy2(int positionX, int positionY, String name): initialize the X and Y coordinates that is inherited from its super class “Enemy” according to given coordinate parameters, and name of the “MinorEnemy2” instances that will be used by “GameObjectFactory” to create these objects.

Methods:

public int getLength():returns the value of length attribute of “MinorEnemy2” object.

public int getWidth():returns the value of width attribute of “MinorEnemy2” object.

public int getVelocity():returns the value of velocity attribute of “MinorEnemy2” object.

public void decrementHealthBy(int diff): this method decrease the “health” of “MinorEnemy2” object by subtracting “diff” parameter which is determined by the damage amount of “LinerRider” which damage the enemy with “LaserCut” powerup or reset to zero by “ClearMinor” powerup.

public int getDamage():returns the value of damage attribute of “MinorEnemy2” object.

public int getHealth():returns the value of health attribute of “MinorEnemy2” object.

6.3. Specifying Contracts

// The length of MajorEnemy1 should be 20 pixel

1) context MajorEnemy1 inv:

    self.LENGTH = 20

// The width of MajorEnemy1 should be 20 pixel

2) context MajorEnemy1 inv:

    self.WIDTH = 10

// The damage on the LineRider that MajorEnemy1 cause should be 5

3) context MajorEnemy1 inv:

    self.DAMAGE = 5

4)

// To decrement the number of lives of LineRider, LineRider's no of lives should be greater
// than 0 and its health should be smaller than zero since before decreasing number of
// lives, health should be consumed.

context LineRider::decrementNoOfLives() pre:

    getNoOfLives() > 0 and

        getHealth() < 0

// After calling decrementNoOfLives() method, the number of lives of LineRider should be
// decreased by 1.

context LineRider::decrementNoOfLives() post:

    getNoOfLives () = @pre. getNoOfLives () - 1

5)

```
// Before cutting the background total area player cutted should  be greater than or equal
// to 0 and less than 80 since if total area is greater or equal to 80 it means that the player
// has already finished the game.
context BackgroundManager::cutBackground() pre:

    getTotalCutRatio () >= 0 and

            getTotalCutRatio() < 80

 // Before cutting the background the player should finish(line should start at an edge point
// and finish at an other edge point ) the cut line he/she draws which isCutProper() method
// check
context BackgroundManager::cutBackground() pre:

    isCutProper()

// After cutting the background, total cut area should be pre total cut ratio minus current
//cut ratio and the line the player draws to cut the area should be destroyed.
context BackgroundManager::cutBackground() post:

    getTotalCutRatio () = @pre.getTotalCutRatio() + getCurrentCutRatio() and

            destroyLine()
```

6)

```
// Before decrementing the health of enemy caused by lineRider, the value obtained by
// subtracting decrease amount from current health of the enemy should be greater than 0
// since otherwise the enemy will be dead.
context Enemy::decreamentHealthBy(h) pre:

    getHealth() – h > 0

// After calling decreamentHealthBy (h) method the health of the enemy decremented by
// parameter h, health decrease amount.

context Enemy::decreamentHealthBy(h) post:


    getHealth() = @pre.getHealth() – h
```

// After implementing the score of the player new score should be pre score plus the
// parameter "s" which is the score gained by user.

7) context GameEngine::incrementScoreBy(s) post:

getScore() = @pre.getScore() + s

8)

// When the position is updated the parameters should be added to pre position values to
// obtain new position values

context Enemy::updatePosition(x,y) post:

getPositionX() = @pre.getPositionX() + x and

getPositionY() = @pre.getPositionY() + y

// When the position is updated the new position should not exceed the size of current
// panel

context Enemy::updatePosition(x,y) post:

getPositionX() < gameEngine.getBackground().getPixels().length and

getPositionY() < gameEngine.getBackground().getPixels()[0].length

// After calling the incrementVelocityBy(v) method the velocity of the LineRider should be
// increased by parameter v, amount of increase velocity.

9) context LineRider::incrementVelocityBy(v) post:

getVelocity() = @pre.getVelocity() + v

// The damage amount of Laser should be 15

10) context Laser inv:

self.DAMAGE = 15

// The duration amount of Laser which is the time LineRider can use this power up should be 5

11) context Laser inv:

self.DURATION = 5

// The velocity of Speed which is the amount that will be added to LineRider's velocity is 3

12) context Speed inv:

self.VELOCITY = 3

13)

// The currentLevel that will be saved initalized to -1
context FileManagerForSave::setCurrentLevel(s) pre:

self.currentLevel = -1

// The currentLevel should be assigned to the level number "s" that is saved
context FileManagerForSave::setCurrentLevel(s) post:

self.currentLevel = s

14)

// After calling the exceedCutRatio () method, the player should totally cut %80 of the field.
context BackgroundManager::exceedCutRatio() pre:

getTotalCutRatio() > 80

// After calling the exceedCutRatio () method the current level number that user plays
// should be increased by 1, since the user passed the level.

context BackgroundManager::exceedCutRatio() post:

gameEngine.currentLevelNo = @pre.gameEngine.currentLevelNo + 1

// The constant properties of GameObjects such as length, width, and damage should be predetermined.

15) context MajorEnemy2 inv:

self.LENGTH = 35 and

self.WIDTH = 5 and

self.DAMAGE = 3

16) context MinorEnemy1 inv:

self.damage = 1 and

length = 3 and

width = 1

17) context MinorEnemy2 inv:

self.damage = 1 and

length = 2 and

width = 2

18)

// Before pausing the game timer should be started which means player plays the game

context GameEngine::pause() pre:

timer. scheduledExecutionTime() > 0

// After pausing the game timer should be stop so that the player can continue from where

// he/she left

context GameEngine::pause() post:

timer. cancel()

// The names of the GameObjects should be predetermined to be called by GameObjectFactory

19) context Speed inv:

self.name = “Speed”

20) context Laser inv:

self.name = “Laser”

21) context ClearMinor inv:

self.name = “ClearMinors”

// The LineRider can have maximum 3 lives and a health range between [0,100]

22) context LineRider inv:

self.getNoOfLives() <= 3 and

self.getHealth() <= 100

// The speed of lineRider should be a nonnegative integer

23) context LineRider inv:

self.getVelocity() >= 0

// Since the total level number is 10 last level can be maximum 9 index starting from 0

24) context FileManagerForLoad inv:

self.lastLevel < 10

// Before updating the object, collision between enemies and LineRider should be checked

// because if there is collision the states of objects will change.

25) context GameEngine::updateObjects() pre:

not CheckCollision()

// Current panel of GameScreenManager should not be null, since it provides the user interaction

26) context GameScreenManager inv:

currentPanel <> null

// The width of the game frame should be 1300 pixel

27) context GameEngine inv:

self.getSize().width = 1300

// The height of the game frame should be 1300 pixel

28) context GameEngine inv:

self.getSize().height = 700

// Since the total level number is 10 current level can be maximum 9 index starting from 0

29) context GameEngine inv:

currentLevelNo < 10

// The duration amount of PauseShieldDecrement which is the time LineRider can use this power
// up should be 10

30) context PauseShieldDecrement inv:

self.DURATION = 10

// X and Y positions of Enemy should be greater than or equal to 0 since java coordinates are
// nonnegative

31) context Enemy inv:

getPositionX() >= 0 and

getPositionY() >= 0

7. Conclusions and Lessons Learned

7.1. Summary

Split-Field is a single player arcade game that undertaken for entertainment. It has unique levels consists of variation of game objects such as enemies, power ups and a line rider. Line rider is the player character whose actions are directly controlled by the player. Enemies are the game objects that damage line rider and block line riders draw a line to specify the boundaries of the cutting area. Power ups are objects that instantly enhance the abilities of line rider when they are cut by line rider. The goal of the game is that line rider should cut the at least 80 percent of the game field to pass the level and finally complete all levels without losing all its lives.

Development process of this game consist of three main stages which are analysis, system design and object design respectively. In analysis design, initially we derive functional and nonfunctional requirements of the system to identify the purpose of the game. Then we have examined possible scenarios that player could perform during the lifecycle of the game which assists us to determine boundary between inside and outside(actor) of the system. Based on these scenarios we have derived use case diagrams. Then we have determined navigational path diagram and user interface of the game that will help to create a better user experience for the player as well as helping creation of distinction between lifecycles of the system. The we have derived the object models and dynamic models to analyze system. Object models contain the domain lexicon which is a glossary for the special terms we have used throughout report and class diagrams which contains classes, functions and relations between them will form the structure of implementation of our project. Dynamic models contain state chart, which gives an abstract description of behavior of the system, and sequence diagram which shows the interaction between objects. By determining these models we have reduced 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.

In the system design report, initially we have determined our design goals based on nonfunctional requirements we have determined in our analysis report. By determining design goals, we aim our system to be reliable, extensible, easy to implement. Secondly, we have decomposed our system to subsystems based on uses cases and analysis models of the system and as the start point, we have used MVC architecture. We have addressed system-wide issues and determined three main subsystem that can be realized independently and encapsulates the states and behaviors of its contained class, which are Game Entities, User Interface, and Game Management. Thirdly, we have determined our architectural patterns as hierarchical layers and MVC architecture. Fourthly, we have mapped our hardware/software system. Finally, we have addressed key concerns such as persistent data management, access control and security, global software control, and boundary conditions to prevent the system from possible problems stem from these concerns.

In the final report, we have decided where to put operations in the object model by adding details to requirements analysis and making implementation decisions under the name of object design. Thus object design serves as the basis of implementation. In object design, initially we have chose three suitable design pattern, which are facade(structural), template method(behavioral), and factory method(creational) pattern, for the system design. Secondly, we have described classes in our design and their public interfaces such as visibility and type signatures in addition to aim and usage of attributes, constructors. Finally, we have specified and explained contracts to specify constraints that cannot otherwise be expressed in a diagram.

7.2. Lessons Learned

In each report, we have learnt different aspects of software engineering. Firstly, from analysis report which constitutes a base step into software engineering we have learned how to describe a problem or specifications of the system and creating scenarios from these specifications. Then, we have learnt how to extract use cases from these scenarios which is an important and valuable requirement analysis technique. We have researched how to make more user-friendly interface that will ease the control of the user on the system. By drawing class, activity, sequence, state chart diagrams we have learned how to apply the theoretical concepts on a real project. To draw these diagrams, we have learnt how to use tools such as Visual Paradigm.

Secondly, from design report we have learnt how to determine design goals in addition to the connection between nonfunctional requirements and design goals. While decomposing the system into subsystems, we have learnt the criteria which we should consider to provide extendibility, hierarchy between classes. By searching for architectural patterns that are suitable to our system, we have learnt which patterns are more suitable to certain project and we had more insight about the patterns we have chose for our system such as MVC and hierarchical layer. In each part of the report, we noticed that we should critically think about each part and make it simple as possible since each change in report may have a significant effect in other parts of the report and in a complex system it would be much harder to detect these effects. For that purpose, we should address key concerns that could cause a significant problem in the future if it is not considered.

Thirdly, from our final report which focuses on the object design, we have learnt how to choose design patterns that are most suitable to our design and realize that in object design we obtain a more concrete system to implement the project. In class interface part, by describing classes, we see that inadequacies in the classes are revealed and the associations between classes

can be observed more concretely. With OCL, we have learnt how to express constraints on UML models.

Overall, from these three reports we have learnt how to apply the theoretical concepts we have learned in class to a real-world project and have a insight about the software engineering process. We see that in order to produce a well-worked project, software engineering is a necessity since it ease the implementation and provides connection between people involved in a project.

7.3. Problems Encountered

First problem we encountered was how to use Visual Paradigm to draw diagrams but after a few unsuccessful trial we have learnt how to use it.

Second problem we have encountered was applying MVC architectural pattern since at first we could not understand how to connect view to controller. Then we could be able to solve how to provide connection between button objects within view class and controller class.

The third problem we have encountered was choosing the appropriate pattern for the our design since there are many patterns with different context and understanding which one is more suitable to our system was requiring effort. By eliminating the unsuitable patterns by looking their context such as for web-based application and structure we have found suitable methods. But we need to change the structure of our class diagram to apply these patterns to our system and especially in MVC part it was hard to change the model, view, controller classes because they constitute the main architecture of the system .

The forth problem we have encountered was become clear when we have started to implement the project because we have realized that there are some missing attributes and

methods in the reports, so we changed the classes slightly in final report according to implementation code.

7.4. Future Work

Since Split-Field game is an extensible, reliable system new features can be added to develop it. A game editor can be added which enable the player to make its own map by constructing game field with the desired game objects he/she want and create level sets. With a game editor user will have more flexibility since he/she will have control on the design process in addition to player character. To make the game more challenging new game objects can be added with more interesting power up or damage properties. Moreover, two player option can be added to make the game more entertaining.