Bilkent University

Department of Computer Engineering

CS 319 Term Project

Section 2

Group 2I

A-day-in-Bilkent

DESIGN Report

Project Group Members

1-Enes Varol

2-Mahammad Askari Iqbal

3-Asaf Kağan Bezgin

4-Alper Kılıçaslan

5-Imran Hajiyev

Supervisor: Uğur Doğrusöz

Contents

1. Introduction

1.1 Purpose of the System

A Day in Bilkent is a shoot 'em' up style bullet hell game. Main purpose of the design is to make the game more enjoyable, and more challenging. The game is based on Bilkent. All the enemies, projectiles, companions are related to Bilkent. Player, which is a student in the game, has to overcome the quizzes, labs, and other assignments to reach the boss. After killing the boss, player can move to the next level. Our aims are improve player's hand-eye coordination, fast decision making, making the player more challenging, and create an enjoyable game. For the coding part, we will use JavaFX library.

1.2 Design Goals

1.2.1 Goals

Game Performance: We want our game as smooth as possible, because everything depends on movement in our game. So that game will

be designed to conserve the frame rate. Hence user can play it easily. Moreover, we want to implement our game so that even outdated computers can run it easily.

User-Friendly Interface: The interface of the game will be simple. Player does not spend extra time to understand the game. Player can access the functions of the game easily so that it will create a user friendly environment. During the game run, player can see the health, score, and other components without breaking the concentration.

Extendibility: The game itself is open for additional extensions. Those extensions will improve the game and the gameplay experience. Changes in the companions, enemies, play modes, etc. will develop the players enthusiasm.

Responsiveness: Goal of the responsiveness is important for both us and players. We want to maintain the frame rate so that players can play the game without disturbance, because lag causes drop in concentration. Hence players do not get the joy of the game. Moreover, buttons will be implemented such a way that players do not get confused.

Animations: Animations in the game is important for the player, because the game is fast and player must see whether bullet touched the enemy. So that, player can come up with new strategies and they can increase their high score.

1.2.2 Trade-offs

1.2.2.1 Functionality vs. Understandability

A Day in Bilkent depends on the abilities, characters, power ups, etc. so that functionality is enough to play the game. However, lots of functions force the players to learn all the functions of the game. We tried to make user friendly as much as we can. If the player knows the descriptions of the elements of the game, one can play better than others.

1.2.2.2 Space vs. Speed

Our game mostly depends on speed, so that we will use as much memory as we need. Game itself should be faster in order to maintain the competitiveness.

1.2.2.3 Rapid Development vs. Functionality

The game has lots of functionality to ease the gameplay. Thus, we have to code all of the functions, but functions take time so if we want to develop the game components it will take more time. However, varieties of the functions make the game more playable and fun. Hence players will enjoy the game.

1.2.2.4 Programmability vs. Speed and Memory

In game implementation, we focus on the speed, because speed is the keyword of our game. In order to increase the speed of the game, we have to use as much memory as we can, so that we do not push CPU to its limits, and everything will work as intended. However, to

maintain the speed, we will code a lot. Hence, the program will be hard to change, because there will be lots of dependencies.

1.2.3 Definitions, Acronyms, Abbreviations

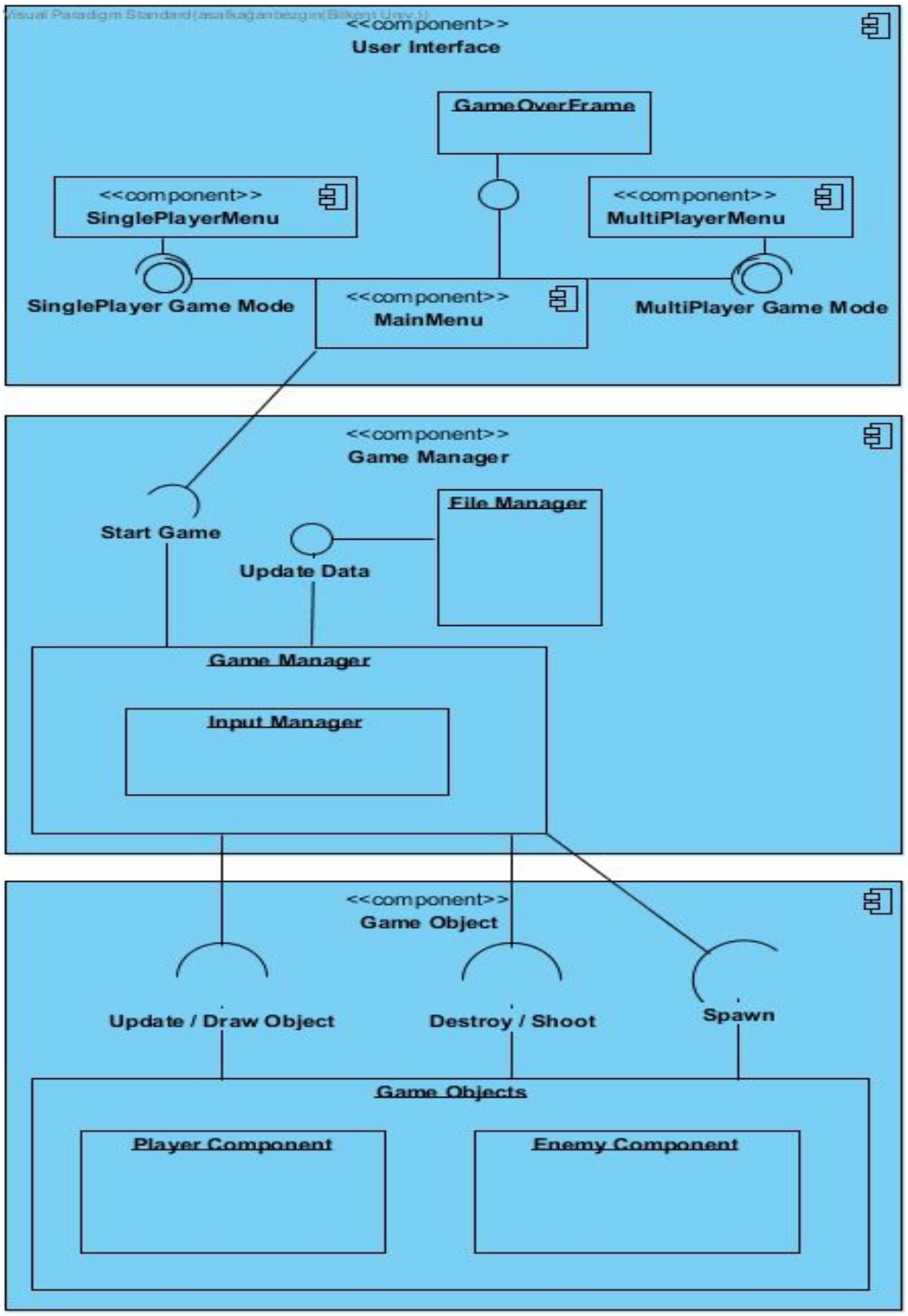
Java Virtual Machine - JVM

Model View Controller - MVC

Graphic User Interface - GUI

2. System Architecture

2.1 Subsystem Decomposition



In this section, the system will be decomposed into maintainable subsystems. The coupling between different subsystems of the main system is reduced and coherence of the components is increased. With the decomposition of the system into different subsystems, it is easier to modify or extend the game when it is necessary.

During the decomposition of the system, MVC is decided to be the best system design pattern to apply on our game. So the system is divided according to MVC principles. (See Figure-1.)

MVC is a suitable design pattern because of the following reasons:

A Day in Bilkent has three subsystem which could be represented with MVC.

- UserInterface Subsystem represents View including user interface components and provides those to user.
- Listeners represents Controller which are keyboard inputs provided from user.
- GameManager and GameObject represent Model which includes every game object.

Our dependencies and relationships between classes can be easily represented by MVC design pattern.

MVC is a simple and effective design pattern which will increment the efficiency of the implement stage.

The system is decomposed into three different subsystems as mentioned before and the *Figure-1* illustrates the decomposed system. Those three subsystems

are User Interface, Listeners, Game Manager & Game Object. The system is devided in those specific subsystems according to their different functionalities. Each subsystem is going to call another subsystem in order to maintain the game. This design helps producers solve any errors that may occur easily and it makes the game more stabilized, modifiable & extendable.

Model Subsystem, includes components for the game entities which are basically game objects. User interface is updated according to the data coming from controller.

Controller is simply the keyboard listeners taking input from the user.

GameManager class access game objects via GameObject class. Same idea also between user interface and GameManager classes. This is called Façade design pattern.

Decomposition of the system into subsystems is going to provide high cohesion and low coupling which will make "A Day In Bilkent" more extendable & flexible.

2.2 Hardware/Software Mapping

Java programming language is going to be used while implementing "A Day In Bilkent" and JAVAFX libraries is going to be used during the process. Since the game is going to be developed by using java programming language, "A Day in Bilkent"

needs to have Java Runtime Environment for the users to execute the game. Since JAVAFX libraries is going to be used, Java Development Kit 8 or newer version.

Keyboard is required as an hardware in order to play the game. It is going to be the I/O tool of the game. The user is going to use keyboard to move around and use items. System requirements of the game is minimal.

The game is going to be stored in .txt files. Highscore, coins, characters, items and companions will be saved in those files. "A Day in Bilkent" is not going to have a database system or internet connection.

2.3 Persistant Data Management

"A Day in Bilkent" is not going to have internet connection or database as mentioned before therefore the game instances is going to be saved in the hard drive of the user which means the game instances will be saved in a .txt file and it will be saved in the hard drive. Items are not going to be changable by user from .txt files because user needs to buy them when they earn coins while they play the game. Highscore is going to be unchangable also.

2.4 Access Control and Security

As mentioned before, "A Day in Bilkent" is not going to use internet connection or any database system but it is necessary to implement basic level security in order to protect the information which is saved in .txt files. It is not

acceptable for any user to change his/her game related information. Therefore the files are not going to be accessable by any user.

2.5 Boundary Conditions

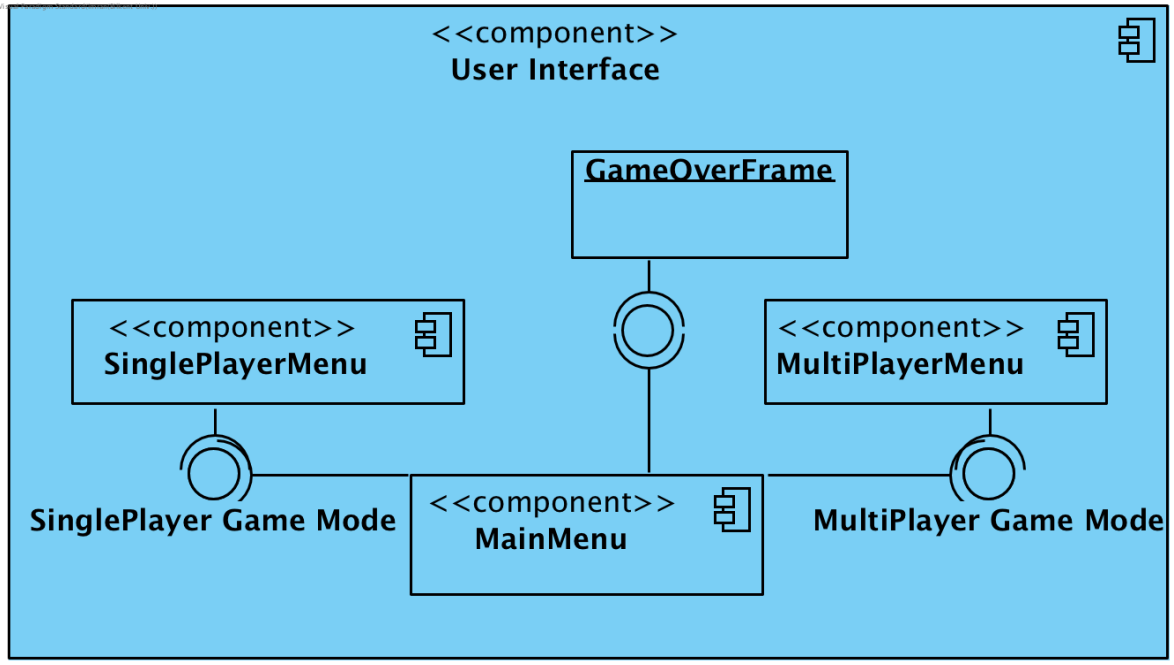
"A Day In Bilkent" is going to be an desktop executable therefore no installation is required. The game file is going to have an .exe extension and a .jar file. The game will be executed from .jar file. This feature is going to bring portability to the game. When the game is carried from one computer to the another, if the computer has necessary software, it will be very easy to run the game.

"A Day In Bilkent" is terminated by using exit button on the menu but there is not going to be a pause button after the game is started.

If an error occurs about images and sounds, the game will be runned without any images and sounds. It can be fixed by changing the necessary game files.

If game stops running due to a performance or implementation problem, the data will be lost.

3. Subsystem Services



3.1 User Interface Subsystem

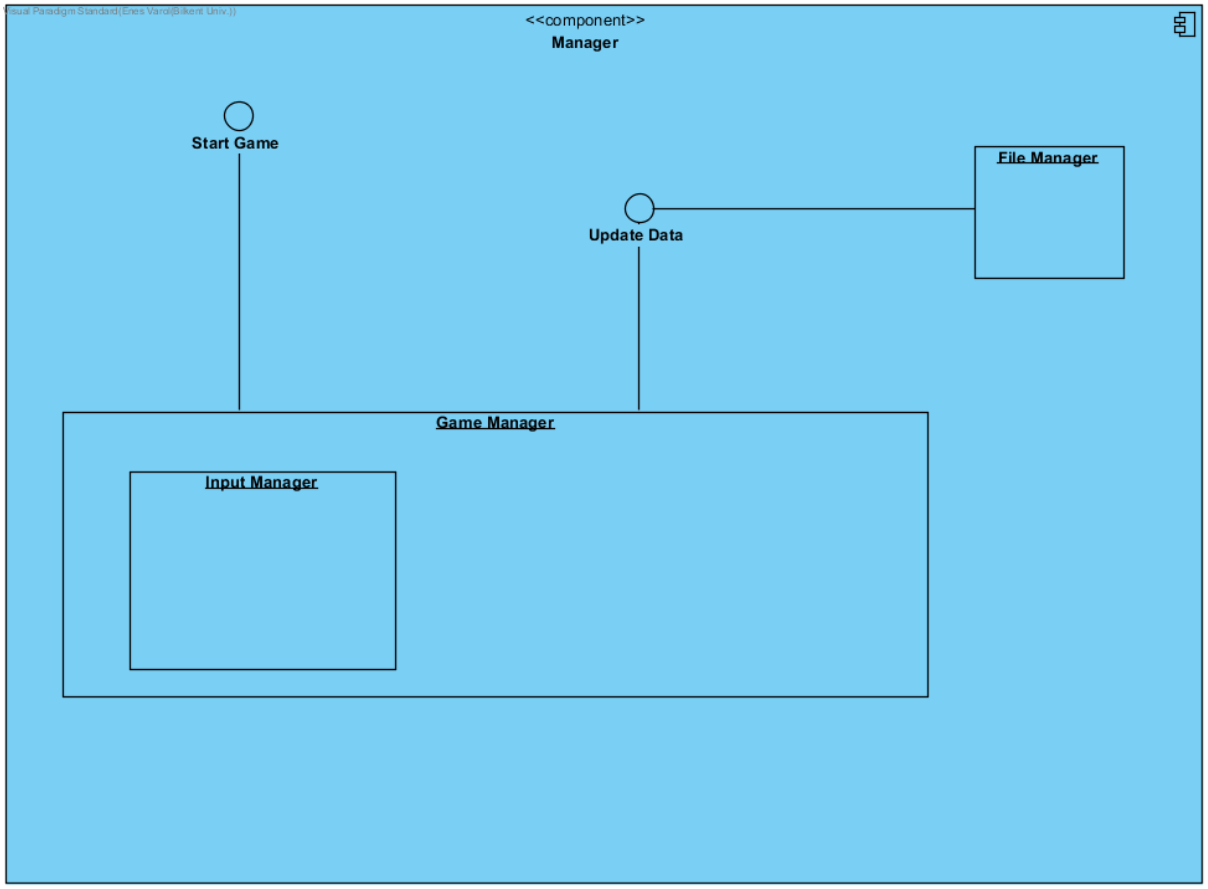
User Interface Subsystem provides the visual part of the game. The high level design of User Interface has three major components and an instance:

1. MainMenu Component
2. SinglePlayerMenu Component
3. MultiPlayerMenu Component
4. GameOverFrame Instance

When the user opens the game, he or she is provided with a menu through the MainMenu component that works together with SinglePlayerMenu and MultiPlayerMenu. MainMenu component has classes for different types of menu options that are common for all the game modes (both single-player and multi-player). SinglePlayerMenu and MultiPlayerMenu have their own functionalities as well, that are prior only to them accordingly. Moreover, menu component can also invoke and instantiate other instance - GameOverFrame instance

that is instantiated when the game ends. Here we can say that Menu Component is the interface which provides different functionalities for the visual user interface.

3.2 Game Manager Subsystem



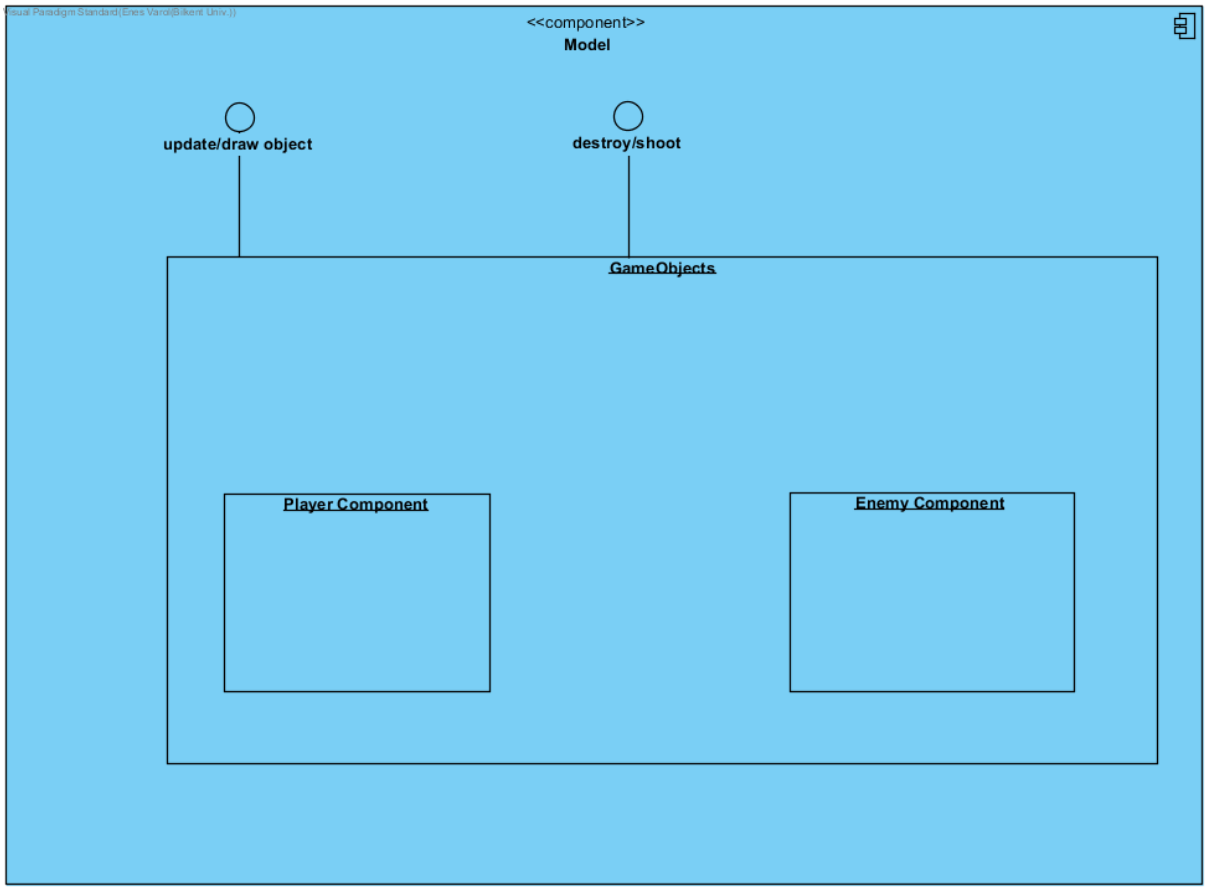
Management Subsystem manages the flow of the game. The high level design of User Interface has three major components:

1. Game Manager
2. File Manager
3. Input Manager

When the game is opened, the manager subsystems controls the responses of the users to visit different pages of the game or change game settings. When player desires to play the game , the game manager subsystem is started. This subsystem is aided by the input manager subsystem that takes in the users input for the game. Using the inputs of the user, the game manager controls the flow of the game. This subsytems works on functionalities such as enemy spawn, wave control , collision detection et cetra that work on the overall

flow of the game. The File Management is used frequently to update the high scores, coins et cetra ; any important thing that needs to be stored for the users future use.

3.3 Model Subsystem



Model Subsystem represents our models. It consists of our Game Objects which can be different types. Model has a main component called GameObjects component which represent different types of game objects/entities. Basically all game entities can only be updated and drawn by the method call coming from GameManagement Subsystem. Other than that destroying objects, checking collisions and creating bullets also part of the model component and can only be called by GameManagement subsystem . Model class has one component which includes two subcomponents:

GameObjects: As mentioned before it represents the model objects.

PlayerComponent: Represents the player and companion, its creates different obstacles and includes their operations.

Enemy Component: Represents the enemies and items/powerUps and includes its operations.

According to the control flow coming from controller model updates it's view by sending calling proper methods which makes changes in GameManagement subsystem.

4. Low-level Design

4.1 Object Designs and Trade-Offs

During implementation we used some of the design patterns. Explanation and the possible conflicts are explained below.

4.1.1 Façade Design Pattern

In our implementation we used façade design pattern between GameObject and GameManager classes. GameManager class access game objects via GameObject class. Same idea also between user interface and GameManager classes.

However, we cannot use Movement, Dimension, Location classes directly. We have to use extra methods. So we use more memory.

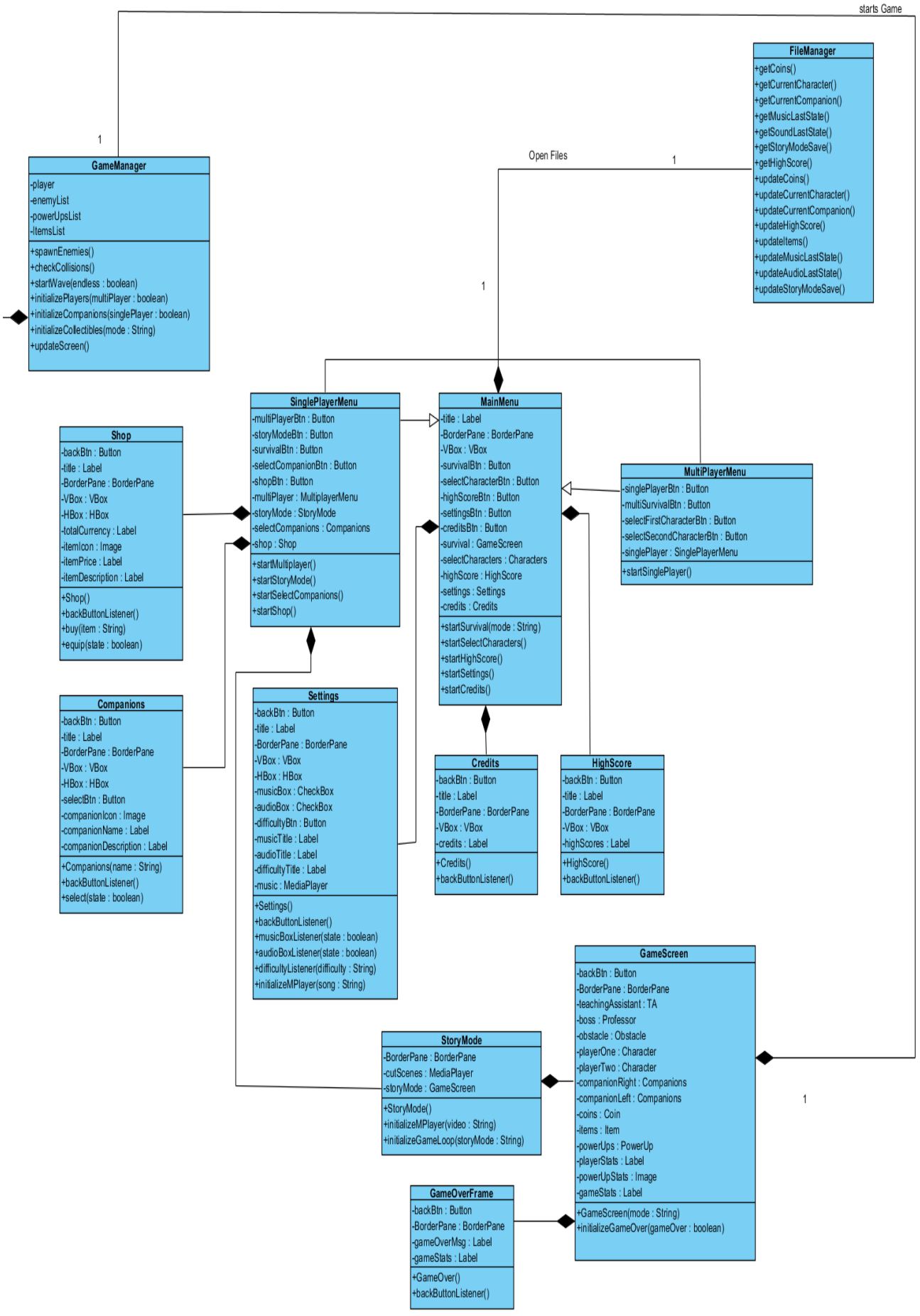
4.1.2 Singleton Design Pattern

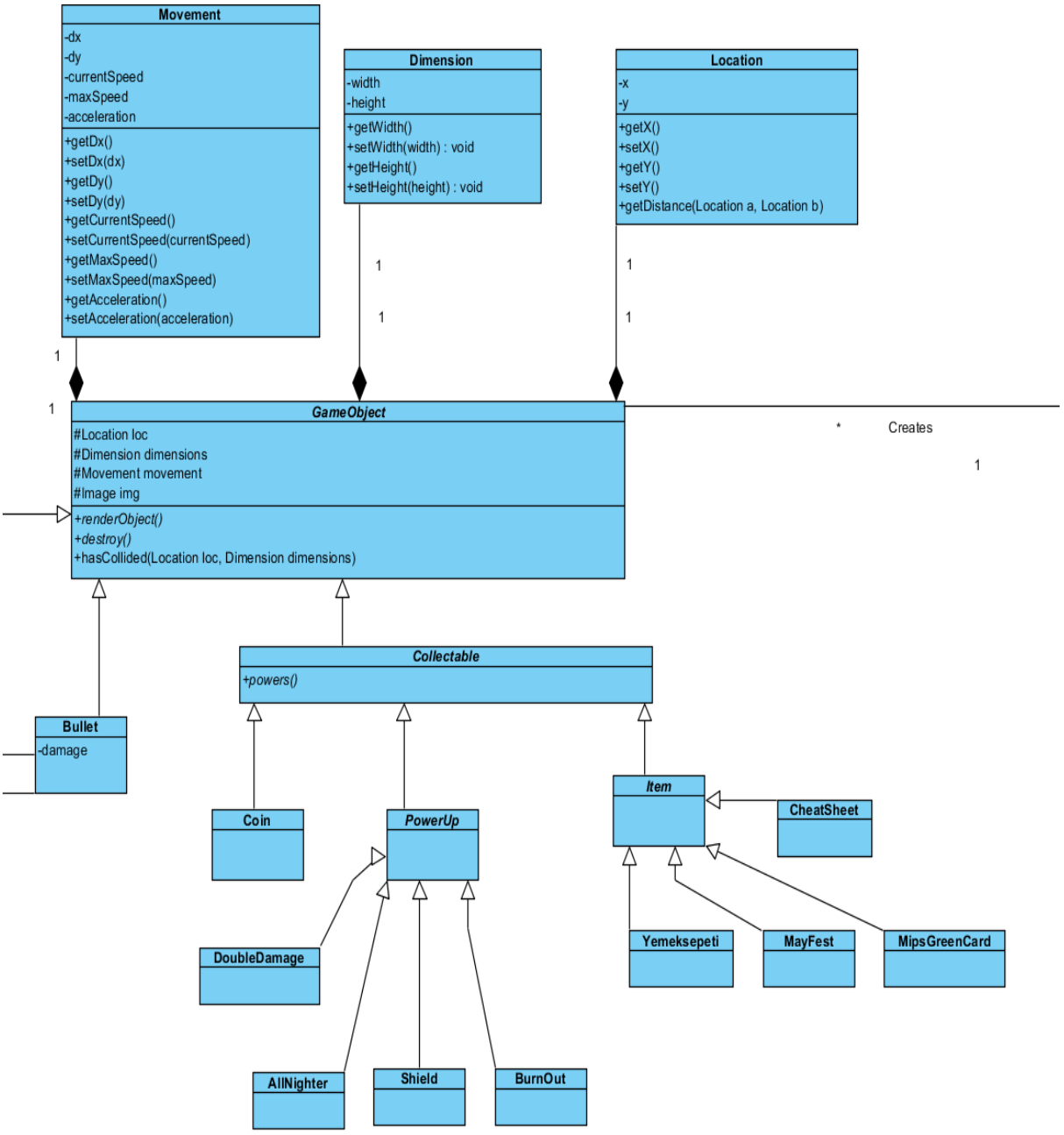
GameManager and FileManager classes have only one instance in MainMenu class so that we prevent any conflict that can happen. So that we maintain the maintainability of the game.

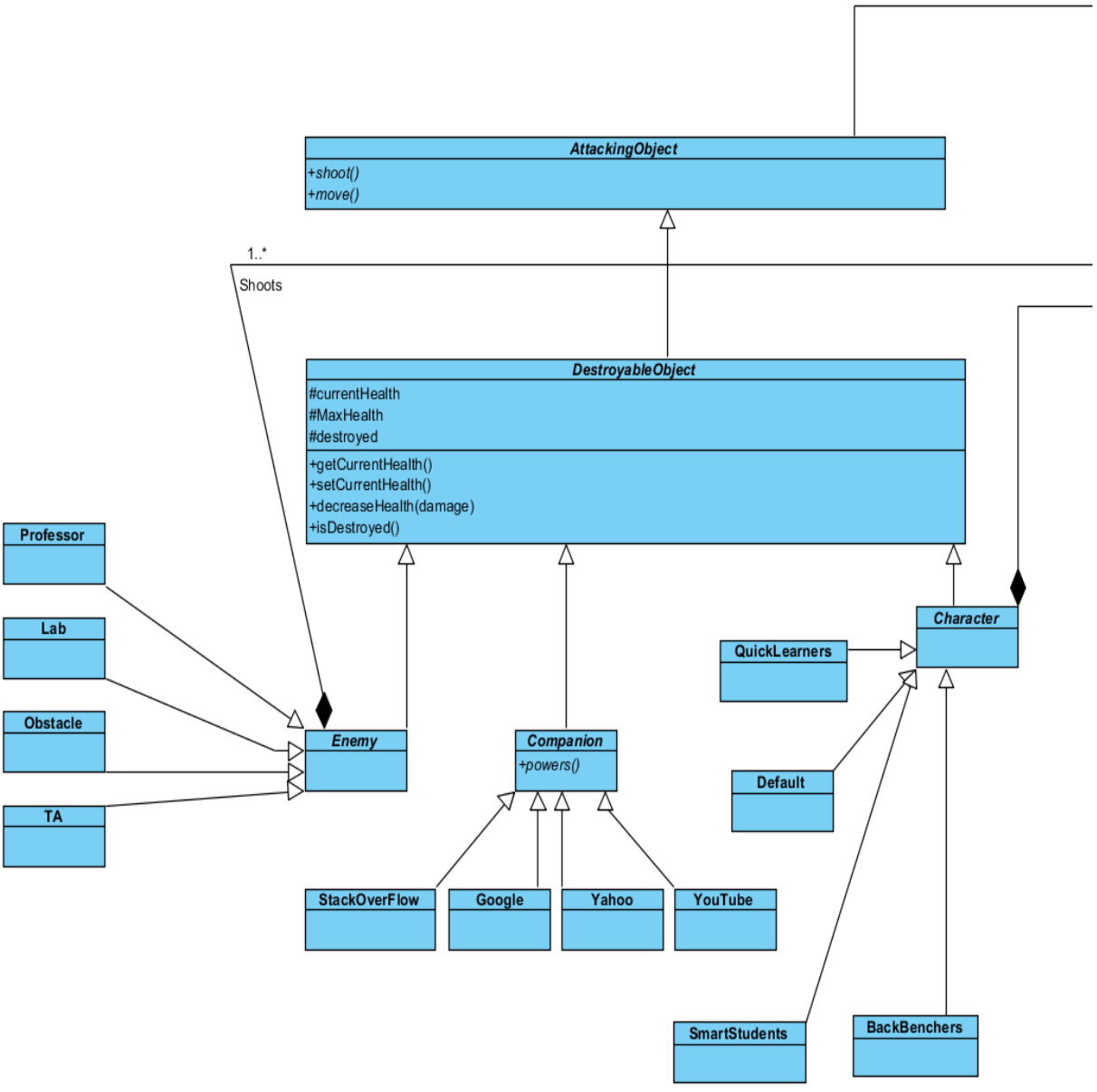
On the other hand, we create static variables so that other classes can access. We have to be careful in order to prevent errors. Unnecessary changes in the static variable could cause errors.

In our implementation, we did not use any Player-Role design pattern.

4.2 Final Object Design




Movement
-dx
-dy
-currentSpeed
-maxSpeed
-acceleration
+getDx()
+setDx(dx)
+getDy()
+setDy(dy)
+getCurrentSpeed()
+setCurrentSpeed(currentSpeed)
+getMaxSpeed()
+setMaxSpeed(maxSpeed)
+getAcceleration()
+setAcceleration(acceleration)
Dimension
-width
-height
+getWidth()
+setWidth(width) : void
+getHeight()
+setHeight(height) : void
Location
-x
-y
+getX()
+setX()
+getY()
+setY()
+getDistance(Location a, Location b)
1
1
1
1
1
1
GameObject
#Location loc
#Dimension dimensions
#Movement movement
#Image img
+renderObject()
+destroy()
+hasCollided(Location loc, Dimension dimensions)
*
Creates
1
Collectable
+powers()
Bullet
-damage
Coin
PowerUp
Item
CheatSheet
DoubleDamage
Yemeksepeti
MayFest
MipsGreenCard
AllNighter
Shield
BurnOut



Game Manager class connected to GameObject , GameObject connected to AttackingObject and Bullet connected to Enemy and Character.

4.3 Packages

4.3.1 Developer's Packages

Menu Package: This package contains the menu elements and their functionality.

Settings Package: This package contains the classes of settings part. Provides methods for sounds and difficulty of the game.

Game Management Package: This package contains the main game elements, such as game loop. It handles communication between classes.

File Management Package: This package handles the file management.

Game Objects Package: This package includes all the game objects such as coins, power ups, characters, etc.

4.3.2 External Packages

java.util: This package will be used for the basic components of the coding such as ArrayList, random number generator, etc., which will help to coding part.

javafx.scene.events: With this package, any events will be examined, and processed according to the design**.**

javafx.scene.input: With this package, input will be taken from the player.

javafx.scene.image: With this package, pictures can be used in the program. This will change the appearance in a fancy way.

javafx.scene.layout: With this package, graphical user interface will be organized as we want to.

javafx.scene.paint: With this package, colors and gradients will be used according to our design.

javafx.scene.Cursor: With this package, cursor will disappear during the game play.

javafx.animation: With this package, animations of the objects can be controllable.

4.4 Class Interfaces

MainMenu

Attributes

- **private Button survivalBtn -** initializes the Survival Mode Game Loop
- **private Button selectCharacterBtn -** accesses the Select Character Screen
- **private Button highScoreBtn -** accesses the HighScore Screen
- **private Button settingsBtn -** accesses the Settings Screen
- **private Button creditsBtn -** accesses the Credits Screens

Methods:

- **public void startSurvival(String mode) -** method that initializes survival mode. String mode that is passed as a parameter decides whether the survival for one player (singleplayer) or two players (multiplayer) will be initialized.
- **public void startSelectCharacters() -** initializes the Characters Screen - to choose specific character.
- **public void startHighScore() -** initializes the HighScore Screen - to check the high scores
- **public void startSettings() -** initializes the Settings Screen - to make custom changes to the game - difficulty, audio, music.
- **public void startCredits() -** initializes the Credits Screen - to check credits.

SinglePlayerMenu

Attributes

- **private Button multiPlayerBtn -** changes the Game Mode to the Multi-player
- **private Button storyModeBtn -** initializes the Story Mode Game Mode
- **private Button selectCompanionBtn -** accesses the Select Companion Screen
- **private Button shopBtn -** accesses the Shop Screen

Methods:

- **public void startMultiplayer() - i**nitializes the MultiPlayer Game Mode - to play with other human beings
- **public void startStoryMode() -** initializes the Story Game Mode - to play and learn the main story, plot of the game.
- **public void startSelectCompanions() - i**nitializes the Companions Screen - to choose specific companion.
- **public void startShop() -** initializes the Shop Screen - to buy/equip needed items.

MultiPlayerMenu

Attributes

- **private Button singlePlayerBtn -** changes the Game Mode to the Single-player
- **private Button firstCharacterBtn -** accesses the Characters Screen
- **private Button secondCharacterBtn -** accesses the Characters Screen

Methods:

- **public void startSinglePlayer() -** initializes the SinglePlayer Game Mode - to play without other human beings and enjoy alone time in the game.

Characters

Attributes

- **private Button backBtn -** back button, returns the player to the main menu
- **private Button selectBtn -** buttons that selects/unselects any specified character
- **private Image characterIcon -** the icon of the character (applies to all)
- **private Label characterName -** the title that shows the name of the character (applies to all)
- **private Label characterStats -** depicts the description of the character (applies to all)

Methods:

- **public void backButtonListener() -** returns to the MainMenu
- **public void select(boolean state) -** selects or unselects any available character. If the boolean value of state is equal to true, then the character is equipped, otherwise it is not.

Constructor:

- **Character(String name, boolean MultiPlayer) -** passes the name of the character as an indicator to the game loop (which character is selected). Boolean multiplayer; - if it's value is equal to true then the selected character is assigned to second player instance, as the system knows its the multiplayer mode. Otherwise it is assigned to the main - first - player.

Companions

Attributes

- **private Button backBtn -** back button, returns the player to the main menu
- **private Button selectBtn -** buttons that selects/unselects any specified companion
- **private Image companionIcon -** icon of the companion (applies to all)
- **private Label companionDescription -** description text of the companion (applies to all)
- **private Label companionName -** the title that shows the name of the companion (applies to all)

Methods:

- **public void backButtonListener() -** returns to the MainMenu
- **public void select(boolean state) -** selects or unselects any available companion. If the boolean value of state is equal to true, then the companion is equipped, otherwise it is not.

Constructor:

- **Companion(String name) -** passes the name of the companion as an indicator to the game loop (which companion is selected).

Shop

Attributes

- **private Button backBtn -** back button, returns the player to the main menu
- **private Button buyBtn -** buttons that buys any specified item
- **private Button equipBtn -** buttons that equips any specified item
- **private Label totalCurrency -** represents total currency of the user currently
- **private Image itemIcon -** icon of the item (applies to all)
- **private Label itemDescription -** represents description for the item (applies to all)
- **private Label itemPrice -** represents price for the item (applies to all)

Methods:

- **public void backButtonListener() -** returns to the MainMenu
- **public void buy(String item) -** buys any available item according to the parameter string - item name.
- **public void equip(boolean state) -** equips any bought item. If the boolean value of state is equal to true, then the item is equipped, otherwise it is not.

Settings

Attributes

- **private Button backBtn -** back button, returns the player to the main menu
- **private CheckBox musicBox -** checked music is on, unchecked music is muted
- **private CheckBox audioBox -** checked audio is on, unchecked audio is muted
- **private Button difficultyBtn -** 3 buttons together as a group: easy, normal and hard difficulty selector
- **private Label musicTitle -** title for the music check box
- **private Label audioTitle** - title for the audio check box
- **private Label difficultyTitle -** title for the difficulty selector button
- **private MediaPlayer music -** instantiates the Java FX's media player to produce music

Methods:

- **public void backButtonListener() -** returns to the MainMenu
- **public void musicBoxListener(boolean state) -** if the boolean value of the state is true then the music is turned on and playing, otherwise its off
- **public void audioBoxListener(boolean state) -** if the boolean value of the state is true then the audio is turned on in the game loop, otherwise its off
- **public void difficultyListener(String difficulty) -** the difficulty string that is passed as a parameter initialize the difficulty of the game in the game loop accordingly
- **public void initializeMPLayer(String song) -** initializes the media player - passing the song as a string to the input of url reader of media player.

HighScore

Attributes

- **private Button backButton -** back button, returns the player to the main menu
- **private Label highScores -** represents the high scores

Methods:

- **public void backButtonListener() -** returns to the MainMenu

Credits Screen

Attributes

- **private Button backButton -** back button, returns the player to the main menu
- **private Label credits -** represents the credits rolling from bottom to top

Methods:

- **public void backButtonListener() -** returns to the MainMenu

StoryMode

Attributes

- **private MediaPlayer cutScenes -** instantiates the Java FX's media player to produce video as a cutscenes between semesters (waves)

Methods:

- **public void initializeMPLayer(String video) -** initializes the media player - passing the video as a string to the input of url reader of media player
- **public void initializeGameLoop(String storyMode) -** initializes the game loop - story mode - passing the story mode as a string will result in a game loop behaving according to the rules of story mode - finite waves; not survival mode.

GameLoop

Methods:

- **public void initializeGameOver(boolean gameOver) -** if the boolean value of gameOver is true then the game is finished and the game over frame pops up. Otherwise the game continues.

Constructor:

- **GameLoop(String mode) -** initializes the mode of the game loop - story mode/survival mode for one player/ survival mode for two players

GameOver Frame

Attributes

- **private Label gameOverMsg -** the message that is showed when the game is over
- **private Label gameStats -** the stats of the player during the game - score, coins and semesters (waves) passed
- **private Button backButton -** back button, returns the player to the main menu

Methods:

- **public void backButtonListener() -** returns to the MainMenu

GameManager

- **public void updateScreen():** Updates the variable values and graphics of the game objects at regular intervals that are done by using JavaFx library.
- **public void initializePlayers(boolean multiPlayer):** Creates a player in the game if multiplayer is false or two players if multiplayer is true.
- **public void initializeCompanions(boolean singlePlayer):** Creates companions in the game for one or two players depending on the input.
- **public void initializeCollectibles(String mode):** Creates collectablesdependig on the mode of the game.
- **public void spawnEnemies(int type):** Creates specific enemies in the game.
- **public void checkCollisions():** Checks for the collision of game objects by looping through all current objects and checking if they have collided. This method exists in all game objects.
- **public void startWave(boolean endless):** Manages the flow of enemies for survival or story mode game. The boolean endless differentiates between the two.
- **public void renderEndGame():** If player loses, this method is called to stop all rendering or updates and show the stats of the game.

Game Object Subsystem

Movement

- **private dx:**This attribute holds the x coordinate of the object.
- **private dy:**This attribute holds the y coordinate of the object.
- **private currentSpeed:**This attribute holds the current speed of the object.
- **private maxSpeed:**This attribute holds the maximum speed of the object.
- **private acceleration:**This attribute holds the acceleration of the object.

Dimension

- **private width:**This attribute holds the width of the object.
- **private height:**This attribute holds the height of the object.

Location

- **private x:**This attribute holds the x coordinate of the object.
- **private y:**This attribute holds the y coordinate of the object.
- **public getDistance(Location a, Location b):**This method returns the distance between two game objects and using their locations as a parameter. Simple geometric formula, distance between two points will be used.

GameObject

- **protected Location loc:**This attribute is a instance of the location class.
- **protected Dimension dimensions:**This attribute is a instance of the dimension class.
- **protected Movement movement:**This attribute is a instance of the movement class.
- **protected Image img:**This attribute holds the image of the game object.

- **public renderObject():**This method creates the image of the object on the screen.
- **public destroy():**This method destroys the image of the object from the screen.
- **public hasCollided(Location loc, Dimension dimensions):**This method checks whether the two game objects has collided or not takes other object's location and dimension as a parameter.

Collectable

- Collectable class extends gameobject class.
- **public powers():**This method activates powers of the collectables and powers affects character stats.

Bullet

- Bullet class extends GameObject class.
- **private damage:** this attribute holds the damage value for the bullet.

AttackingObject

- AttackingObject class extends GameObject class.
- **public shoot():** this method creates bullets on the screen bullets also use circle as shape.
- **public move():** this method updates the object's location gets player's input and changes the location with respect to that.

DestroyableObject

- DestroyableObject class extends AttackingObject class.
- **protected currentHealth:** this attribute contains the current health of the object.
- **protected maxHealth:** this attribute contains the maximum health of the object.
- **protected destroyed:** this attribute contains the destroyed or not situation of the object.
- **public decreaseHealth(damage):**this method deacreses the current health of the object and takes damage as parameter.
- **public isDestroyed():**this method checks object's current health and returns true or false with respect to that.

Companion

- Companion class extends DestroyableObject class.
- **public powers():** this method activates the power of the companion every company has different power like shield or kill all enemies and powers have cooldown times.

5. Improvement Summary

In the second iteration of the design report we have done significant changes. Firstly, because of the additions such as story-mode, local multiplayer we did in the second analysis report, our objects design has changed. Since our object design is affected, accordingly we have added some additional trade-offs for high level design and also for object design trade-offs. We will list the important changes for briefty of the report:

- We have changed our system decomposition. Our main design architectural pattern which is MVC is stayed as the same. However, our subsystems have

changed. In the second iteration rather than writing all of the classes in a subsystem, we have decomposed our system in three main components UserInterface, Listeners, GameManager & GameObject which are equivalent to View, Controller, Model respectively.

- Our representation of subsystem decomposition is totally changed. To show the relationship between subsystems we have used lollipop component diagrams.
- We have decomposed each subsystem to smaller components if possible. The division of components made according to functionalities of different classes. We have changed the names of subsystem that we used in the first iteration.
- In Subsystem decomposition, we explained our subsystems and their relationship with our architectural design pattern which is MVC.
- We have added some design patterns to achieve to have a complete object oriented design. Eg. Singleton Design Pattern, Façade Design Pattern and Player-Role Pattern.
- We have added 'Design Patterns' section in which we have discussed our design patterns more detailed and explained their trade-offs.
- In the low level design, the object class diagram is updated. Its complete version and some pieces are given for more clarity on object model.
- The class interfaces are updated. Class explanations made as clear as possible.
- The packages are updated.

6. Glossary & References

Javafx : https://docs.oracle.com/javase/8/javafx/api/toc.htm

Revised:

Added Manager subsystem.
Added more attributes in the manager classes.
Formatted the document better.