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 Moreover, we want to implement our game so that even outdated computers can run it easily.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.

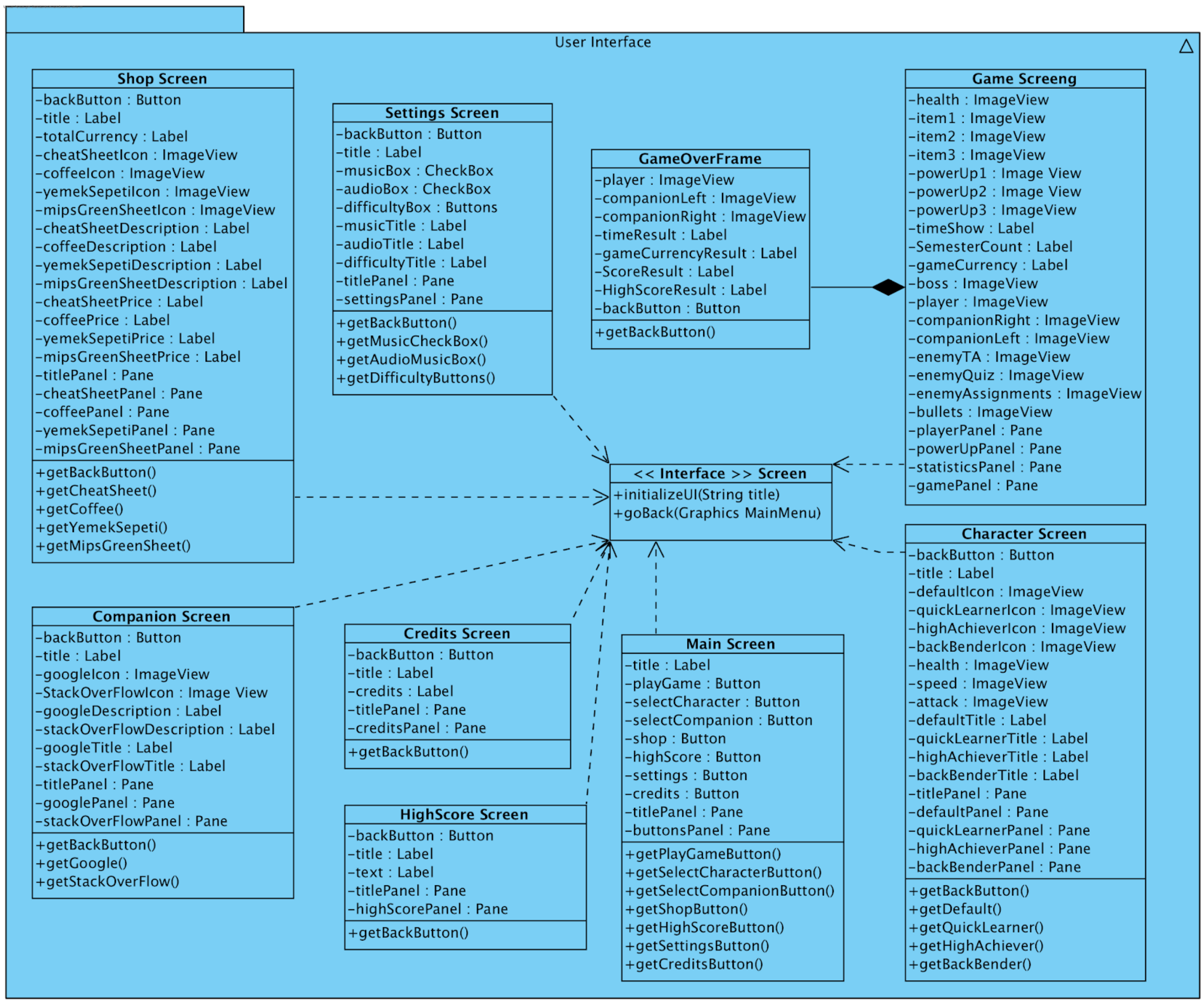
"A Day In Bilkent" is terminated by using exit button on the menu but there is not going to be a pause button after the game is started.

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

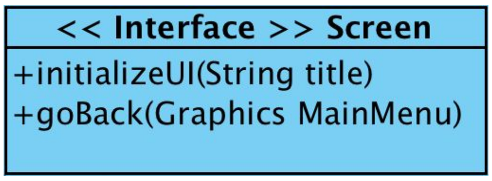


User Interface Subsystem of «A day in Bilkent» is composed of 8 different classes, each for one screen and a Frame that is going to be accessed in one of those 8 classes. All the classes extend the Screen Interface which consists of two methods: initializing the GUI and going back to the Main Menu. Main Menu can be called the main screen for the game, because all the other screens are accessed through it. Each of the 8 screens have their own controller classes, which will implement all the needed functionality. However all GUI classes have get methods in order to give the controller classes the ability to access needed buttons. GameOverFrame is inside the Game Screen Class and will be called only when the player dies or completes the game, that is when the game ends. Overall, the diagram of the User Interface Subsystem above shows how the UI will be implemented and how it functions.

Key points:

1. All Screen Classes are the done using MVC logic and are strictly GUI.
2. All the Screen Classes are implementing the same Screen Interface.
3. The Main Screen is a menu screen which gives access to all the other screens through predefined buttons.
4. Almost all the Screen Classes, except Game Screen, have get methods as a link to their Controller Classes.
5. Game Screen has the GameOverFrame which will be used once the game ends.

<<Interface>> Screen



public void initializeUI(String title) - initializes the needed UI due to the specified parameter. For example: title is Settings -> Settings UI is initialized.

• public void goBack(Graphics MainMenu) - When user returns to Main Menu, loads it.

Main Screen

Attributes

- private Label title - the title of the Screen Class that is currently executing
- private Button playGame - accesses the Game Screen
- private Button selectCharacter - accesses the Select Character Screen
- private Button selectCompanion - accesses the Select Companion Screen
- private Button shop - accesses the Shop Screen
- private Button highScore - accesses the HighScore Screen
- private Button settings - accesses the Settings Screen
- private Button credits - accesses the Credits Screen
- private Panel titlePanel - panel for the title
- private Panel buttonsPanel - panel for 7 predefined buttons

Methods:

- public Button getPlayGameButton() - returns the Play Game button to the controller
- public Button getSelectCharacterButton() - returns the Select Character button to the controller
- public Button getSelectCompanionButton() - returns the Select Companion button to the controller
- public Button getShopButton() - returns the Shop button to the controller
- public Button getHighScoreButton() - returns the HighScore button to the controller
- public Button getSettingsButton() - returns the Settings button to the controller
- public Button getCreditsButton() - returns the Credits button to the controller

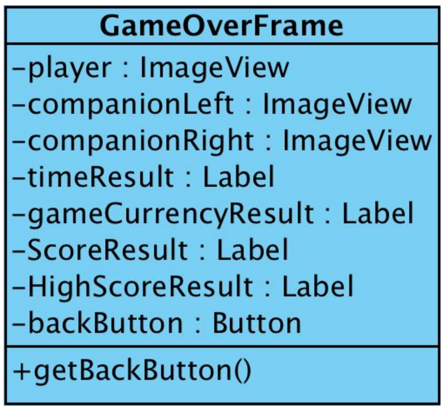
Game Screen

Attributes

- private ImageView health - the current health of the Player
- private ImageView item1 - if equipped, one of the items of the Player
- private ImageView item2 - if equipped, one of the items of the Player
- private ImageView item3 - if equipped, one of the items of the Player
- private ImageView powerUp1 - if equipped, one of the drop-powerUps of the Player
- private ImageView powerUp2 - if equipped, one of the drop-powerUps of the Player
- private ImageView powerUp3 - if equipped, one of the drop-powerUps of the Player
- private ImageView player - the Player model
- private ImageView companionRight - the right Companion model
- private ImageView companionLeft - the left Companion model
- private ImageView boss - the main boss model
- private ImageView enemyTA - model of one of the enemies
- private ImageView enemyQuiz - model of one of the enemies
- private ImageView enemyAssignments - model of one of the enemies
- private ImageView bullets - model of bullets that are shot at player/by player
- private Label timeShow - the time elapsed during the game
- private Label semesterCount - the waves of enemies cleared
- private Label gameCurrency - the game currency gained during the game
- private Pane playerPanel - panel for the player's health and items
- private Pane powerUpPanel - panel for the player's equipped power ups
- private Pane statisticsPanel - panel for the statistics of the game played

- private Pane gamePanel - the main game panel

GameOver Frame



Attributes

- private ImageView player - the Player model
- private ImageView companionRight - the right Companion model
- private ImageView companionLeft - the left Companion model
- private Label timeResult - the final time result of the game played
- private Label gameCurrencyResult - the collected currency
- private Label ScoreResult - the final score the player got
- private Label HighScoreResult - the high score the player has: beaten/unbeaten
- private Button backButton - back button, returns the player to the main menu

Methods:

- public Button getBackButton() - returns the Back button to the controller

Character Screen

Attributes

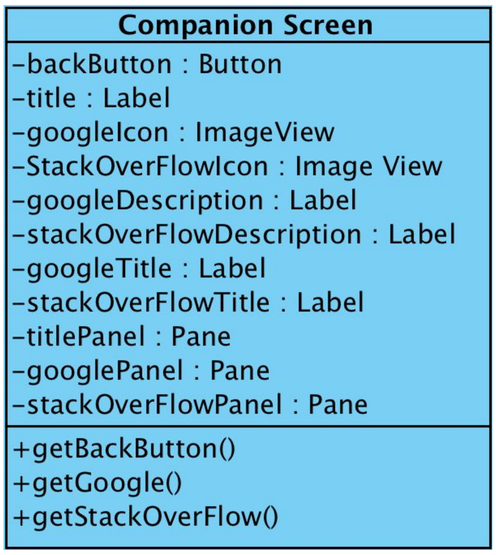
- private Button backButton - back button, returns the player to the main menu
- private Label title - the title of the Screen Class that is currently executing
- private ImageView defaultIcon - the icon of the default character
- private ImageView quickLearnerIcon - the icon of the Quick Learner character
- private ImageView highAchiever - the icon of the highAchiever character
- private ImageView backBenderIcon - the icon of the Back Bender character
- private ImageView health - health of the characters
- private ImageView speed - speed of the characters
- private ImageView attack - attack of the characters
- private Label defaultTitle - the title that shows the name of the default character
- private Label quickLearnerTitle - the title that shows the name of the Quick Learner character
- private Label highAchieverTitle - the title that shows the name of the High Achiever character
- private Label backBenderTitle - the title that shows the name of the Back Bender character
- private Pane titlePanel - panel for the title
- private Pane defaultPanel - panel for the default character, his statistics and icon
- private Pane quickLearnerPanel - panel for the Quick Learner character, his statistics and icon
- private Pane highAchieverPanel - panel for the High Achiever character, his statistics and icon
- private Pane backBenderPanel - panel for the Back Bender character, his statistics and icon

Methods:

- public Button getBackButton() - returns the Back button to the controller
- public Button getDefault() - returns the Default character choice to the controller

- public Button getQuickLearner() - returns the Quick Learner character choice to the controller
- public Button getHighAchiever() - returns the High Achiever character choice to the controller
- public Button getBackBender() - returns the Back Bender character choice to the controller

Companion Screen



Attributes

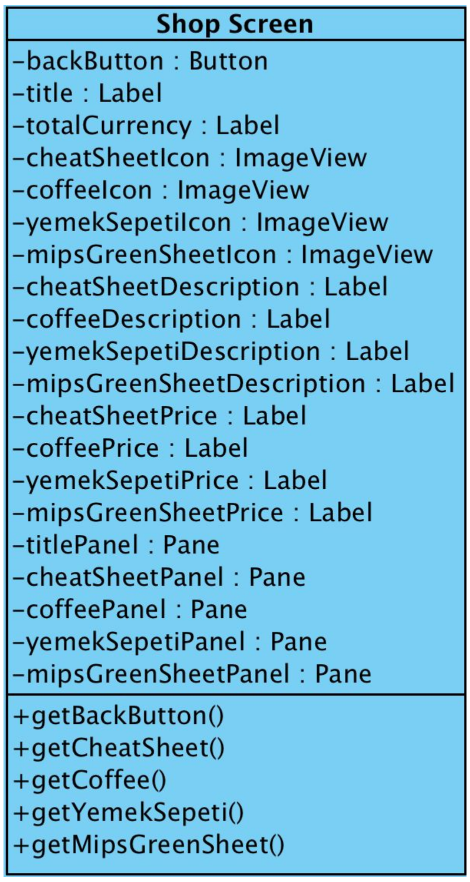
- private Button backButton - back button, returns the player to the main menu
- private Label title - the title of the Screen Class that is currently executing
- private ImageView googleIcon - icon for the Google companion
- private ImageView stackOverFlowIcon - icon for the Stack OverFlow companion
- private Label googleDescription - description text of the Google companion
- private Label stackOverFlowDescription - description text of the Stack OverFlow companion
- private Label googleTitle - the title that shows the name of the Google companion

- private Label stackOverFlowTitle - the title that shows the name of the Stack OverFlow companion
- private Pane titlePanel - panel for the title
- private Pane googlePanel - panel for the Google Companion: icon, title and description
- private Pane stackOverFlowPanel - panel for the Stack OverFlow Companion: icon, title and description

Methods:

- public Button getBackButton() - returns the Back button to the controller
- public Button getGoogle() - returns the Google companion choice to the controller
- public Button getStackOverFlow() - returns the Stack OverFlow companion choice to the controller

- **Shop Screen**



Shop Screen

```
-backButton : Button
-title : Label
-totalCurrency : Label
-cheatSheetIcon : ImageView
-coffeeIcon : ImageView
-yemekSepetiIcon : ImageView
-mipsGreenSheetIcon : ImageView
-cheatSheetDescription : Label
-coffeeDescription : Label
-yemekSepetiDescription : Label
-mipsGreenSheetDescription : Label
-cheatSheetPrice : Label
-coffeePrice : Label
-yemekSepetiPrice : Label
-mipsGreenSheetPrice : Label
-titlePanel : Pane
-cheatSheetPanel : Pane
-coffeePanel : Pane
-yemekSepetiPanel : Pane
-mipsGreenSheetPanel : Pane
+getBackButton()
+getCheatSheet()
+getCoffee()
+getYemekSepeti()
+getMipsGreenSheet()
```

Attributes

- private Button backButton - back button, returns the player to the main menu
- private Label title - the title of the Screen Class that is currently executing
- private Label totalCurrency - represents total currency of the user currently

- private ImageView cheatSheetIcon - icon of the Cheat Sheet item
- private ImageView coffeeIcon - icon of the Coffee item
- private ImageView yemekSepetiIcon - icon of the YemekSepeti item
- private ImageView mipsGreenSheetIcon - icon of the MIPS Green Sheet item
- private Label cheatSheetDescription - represents description for the Cheat Sheet item
- private Label coffeeDescription - represents description for the Coffee item
- private Label yemekSepetiDescription - represents description for the YemekSepeti item
- private Label mipsGreenSheetDescription - represents description for the MIPS Green Sheet item
- private Label cheatSheetPrice - represents price for the Cheat Sheet item
- private Label coffeePrice- represents price for the Coffee item
- private Label yemeksepetiPrice - represents price for the YemekSepeti item
- private Label mipsGreenSheetPrice - represents price for the MIPS Green Sheet item
- private Pane titlePanel - panel for the title
- private Pane cheatSheetPanel - panel for the Cheat Sheet item: icon, title, description, price
- private Pane coffeePanel - panel for the Coffee item: icon, title, description, price
- private Pane yemekSepetiPanel - panel for the YemekSepeti item: icon, title, description, price
- private Pane mipsGreenSheetPanel - panel for the MIPS Green Sheet item: icon, title, description, price

Methods:

- public Button getBackButton() - returns the Back button to the controller
- public Button getCheatSheet() - returns the Cheat Sheet button to the controller
- public Button getCoffee() - returns the Coffee button to the controller
- public Button getYemekSepeti() - returns the YemekSepeti button to the controller
- public Button getMIPSGreenSheet() - returns the MIPS Green Sheet button to the controller

Settings Screen

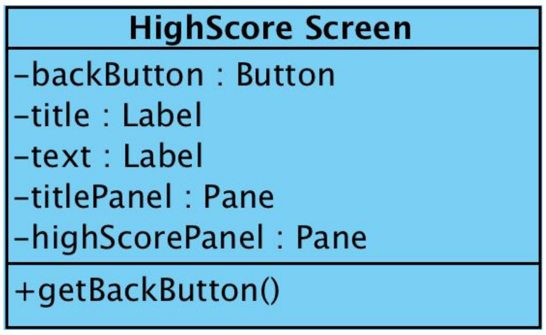
Attributes

- private Button backButton - back button, returns the player to the main menu
- private Label title - the title of the Screen Class that is currently executing
- private CheckBox musicBox - checked music is on, unchecked music is muted
- private CheckBox audioBox - checked audio is on, unchecked audio is muted
- private Button difficultyBox - 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 Pane titlePanel - panel for the title
- private Pane settingsPanel - panel for the settings options, checkboxes and buttons

Methods:

- public Button getBackButton() - returns the Back button to the controller
- public Button getMusicCheckBox() - returns the music box checked/unchecked to the controller
- public Button getAudioCheckBox() - returns the audio box checked/unchecked to the controller
- public Button getDifficultyButtons() - returns the selected difficulty button to the controller

HighScore Screen



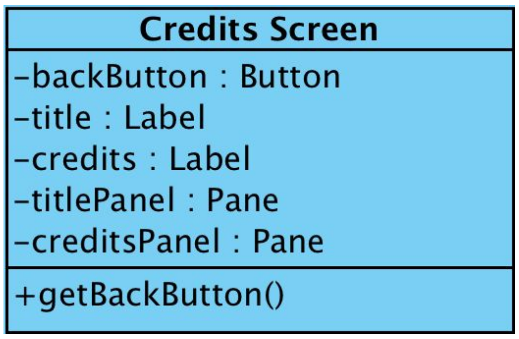
Attributes

- private Button backButton - back button, returns the player to the main menu
- private Label title - the title of the Screen Class that is currently executing
- private Label text - represents the high scores
- private Pane titlePanel - panel for the title
- private Pane highScorePanel - panel for the high scores represented as texts

Methods:

- public Button getBackButton() - returns the Back button to the controller

Credits Screen



Attributes

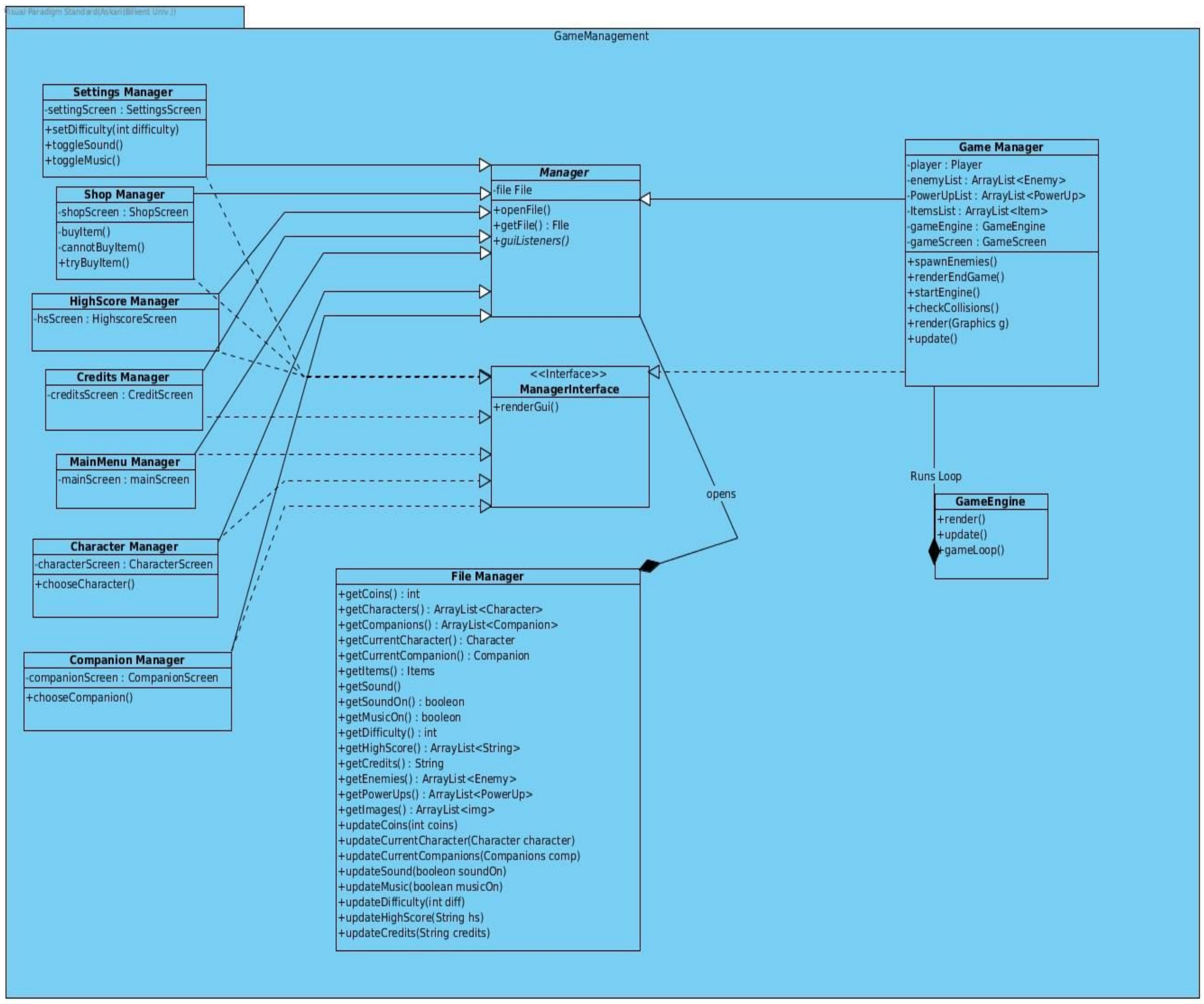
- private Button backButton - back button, returns the player to the main menu
- private Label title - the title of the Screen Class that is currently executing
- private Label credits - represents the credits rolling from bottom to top
- private Pane titlePanel - panel for the title

• private Pane creditsPanel - panel for the credits represented as a text

Methods:

• public Button getBackButton() - returns the Back button to the controller

3.2 Game Management Subsystem



Visual Paradigm Standard(Askari(Bilkent Univ.))
GameManagement
Settings Manager
-settingScreen : SettingsScreen
+setDifficulty(int difficulty)
+toggleSound()
+toggleMusic()
Shop Manager
-shopScreen : ShopScreen
-buyItem()
-cannotBuyItem()
+tryBuyItem()
HighScore Manager
-hsScreen : HighscoreScreen
Credits Manager
-creditsScreen : CreditScreen
MainMenu Manager
-mainScreen : mainScreen
Character Manager
-characterScreen : CharacterScreen
+chooseCharacter()
Companion Manager
-companionScreen : CompanionScreen
+chooseCompanion()
Manager
-file File
+openFile()
+getFile() : FIle
+guiListeners()
<<Interface>>
ManagerInterface
+renderGui()
Game Manager
-player : Player
-enemyList : ArrayList<Enemy>
-PowerUpList : ArrayList<PowerUp>
-ItemsList : ArrayList<Item>
-gameEngine : GameEngine
-gameScreen : GameScreen
+spawnEnemies()
+renderEndGame()
+startEngine()
+checkCollisions()
+render(Graphics g)
+update()
Runs Loop
GameEngine
+render()
+update()
+gameLoop()
opens
File Manager
+getCoins() : int
+getCharacters() : ArrayList<Character>
+getCompanions() : ArrayList<Companion>
+getCurrentCharacter() : Character
+getCurrentCompanion() : Companion
+getItems() : Items
+getSound()
+getSoundOn() : booleon
+getMusicOn() : booleon
+getDifficulty() : int
+getHighScore() : ArrayList<String>
+getCredits() : String
+getEnemies() : ArrayList<Enemy>
+getPowerUps() : ArrayList<PowerUp>
+getImages() : ArrayList<img>
+updateCoins(int coins)
+updateCurrentCharacter(Character character)
+updateCurrentCompanions(Companions comp)
+updateSound(booleon soundOn)
+updateMusic(boolean musicOn)
+updateDifficulty(int diff)
+updateHighScore(String hs)
+updateCredits(String credits)

Figure 3.2 **Detailed User Interface Subsystem**

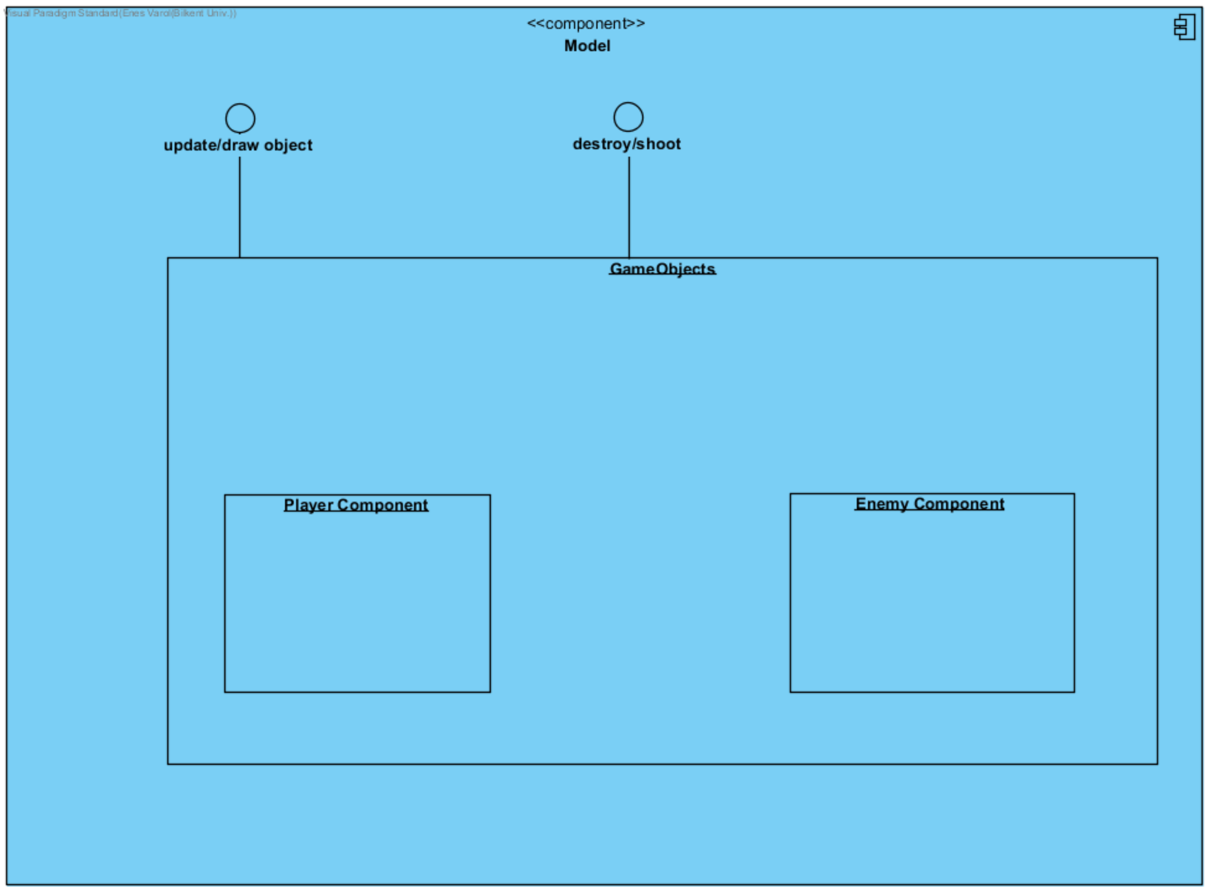
Game Management Subsystem of «A day in Bilkent» contains an interface and one master class called Manager that handles all the common functions of all managers. The managers are controllers for all the screens that are available and have to collect data, listen to button calls and send data back to the file systems. The game manager is unique as it has to run a game loop as well to run the game.

Key points:

1. All Manager Classes are the done using MVC logic and contain both GUI and handling of the GUI.
2. All the Manager Classes are implementing the same Manager Interface.
3. The Main Manager handles all the inputs of the game to other sections.

- Interface enforces the managers to render gui using same method name
- Manager abstract class handles the file managing system and has an abstract method for all managers to implement their own gui listeners
- Managers handles the gui listening problems of their own screens.
- Game Managers has game engine logic , collision logic and handles enemy spawning. It also takes care of game end.

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 Packages

4.2.1 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.

4.2.2 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.

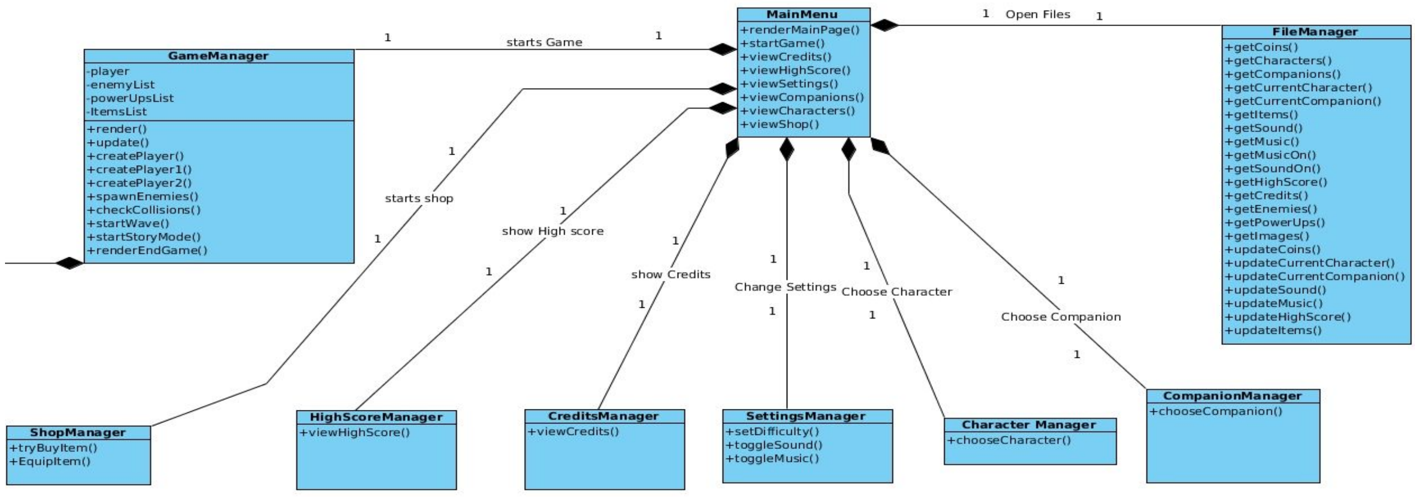
javafx.scene.paint: With this package, colors and gradients will be used according to our design.

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.

4.3 Class Interfaces

4.3.2 Game Manager Subsytem



MainMenu

- **public void renderMainPage(): It renders the main menu by calling the UI file.**
- **public void startGame(): Makes the Game Manager object.**
- **public void viewCredits(): Makes the CreditsManager object.**
- **public void viewHighScore(): Makes the HighScoreManager object.**

- **public void viewSettings(): Makes the SettingsManager object.**
- **public void viewCharacter(): Makes the CharacterManager object.**
- **public void viewCompanion(): Makes the CompanionManager object.**
- **public void viewShop(): Makes the ShopManager object.**

ShopManager

- **public boolean tryBuyItem(Item): Checks the price of the item and returns true or false depending on if the player has enough coins to purchase an item. If they do, the item is added to the players inventory.**
- **public void equipItem(Item): Selects or dis-selects an item for the player.**

HıghScoreManager

- **public void viewHighScore(): It renders the HighScores by calling the UI file**

CreditsManager

- **public void viewCredits(): It renders the Credits by calling the UI file**

SettingsManager

- **public void setDifficulty(int d): It sets the difficulty depending on what the player chooses.**
- **public void toggleSound(): Turns Sound on or off.**
- **public void toggleMusic(): Turns Music on or off.**

CharacterManager

- **public void chooseCharacter(): Sets the current character for player.**

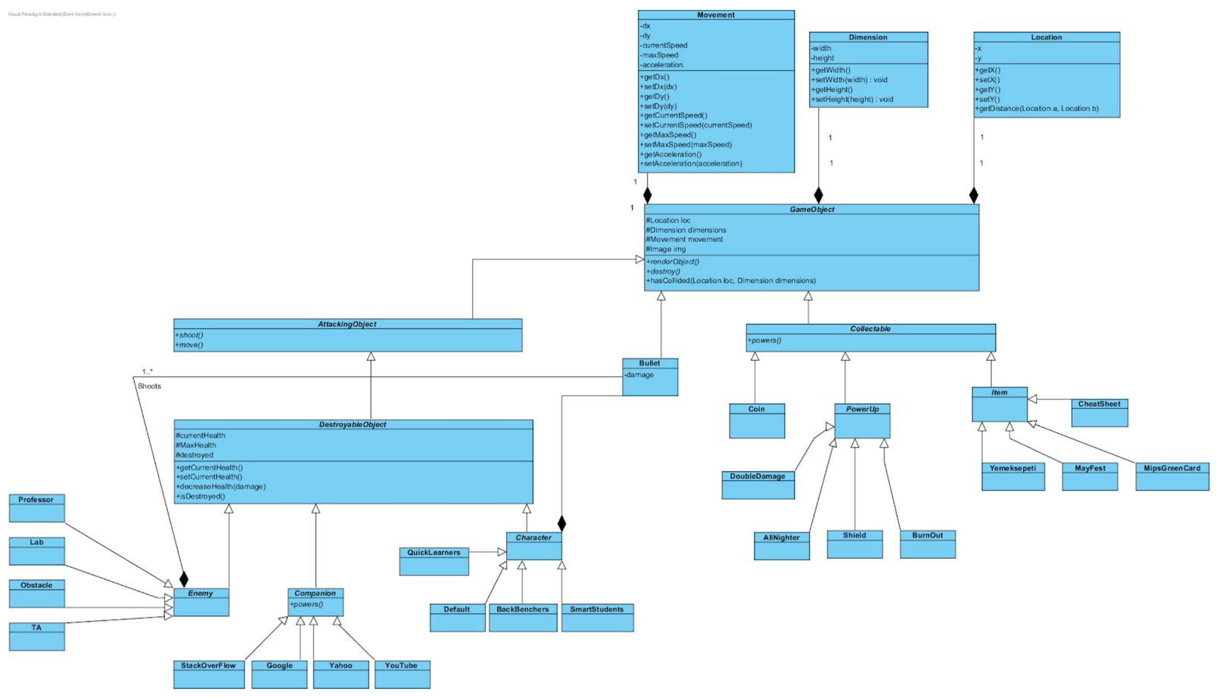
CompanionManager

- **public void chooseCompanion(): Sets the current companion for player.**

GameManager

- **public void render(): Updates the images / graphics of the game at regular intervals that are done by using JavaFx library.**
- **public void update(): Updates the variable values of the game objects at regular intervals that are done by using JavaFx library.**
- **public void createPlayer(): Creates a player in the game.**
- **public void spawnEnemies(): Spawns enemies into 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(): Manages the flow of enemies for a survival game.**
- **public void startStoryMode(): Manages the flow of enemies for a story mode game.**
- **public void renderEndGame(): If player loses, this method is called to stop all rendering or updates and show the stats of the game.**

4.3.3 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