

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. **Purpose of the system**
   2. **Design Goals**

**2. System Architecture**

**2.1 Subsystem Decomposition**

**2.2 Hardware/Software Mapping**

**2.3 Persistent Data Management**

**2.4 Access Control and Security**

**2.5 Boundary Conditions**

**3. Subsystem Services**

**3.1 User Interface Subsystem**

**3.2 Game Manager Subsystem**

**3.3 Game Objects Subsystem**

**4. Low-level Design**

**4.1 Object Design Trade-Offs**

**4.2 Packages**

**4.3 Class Interfaces**

**5. Glossary & References**

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 Moreover, we want to implement our game so that even outdated computers can run it easily.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

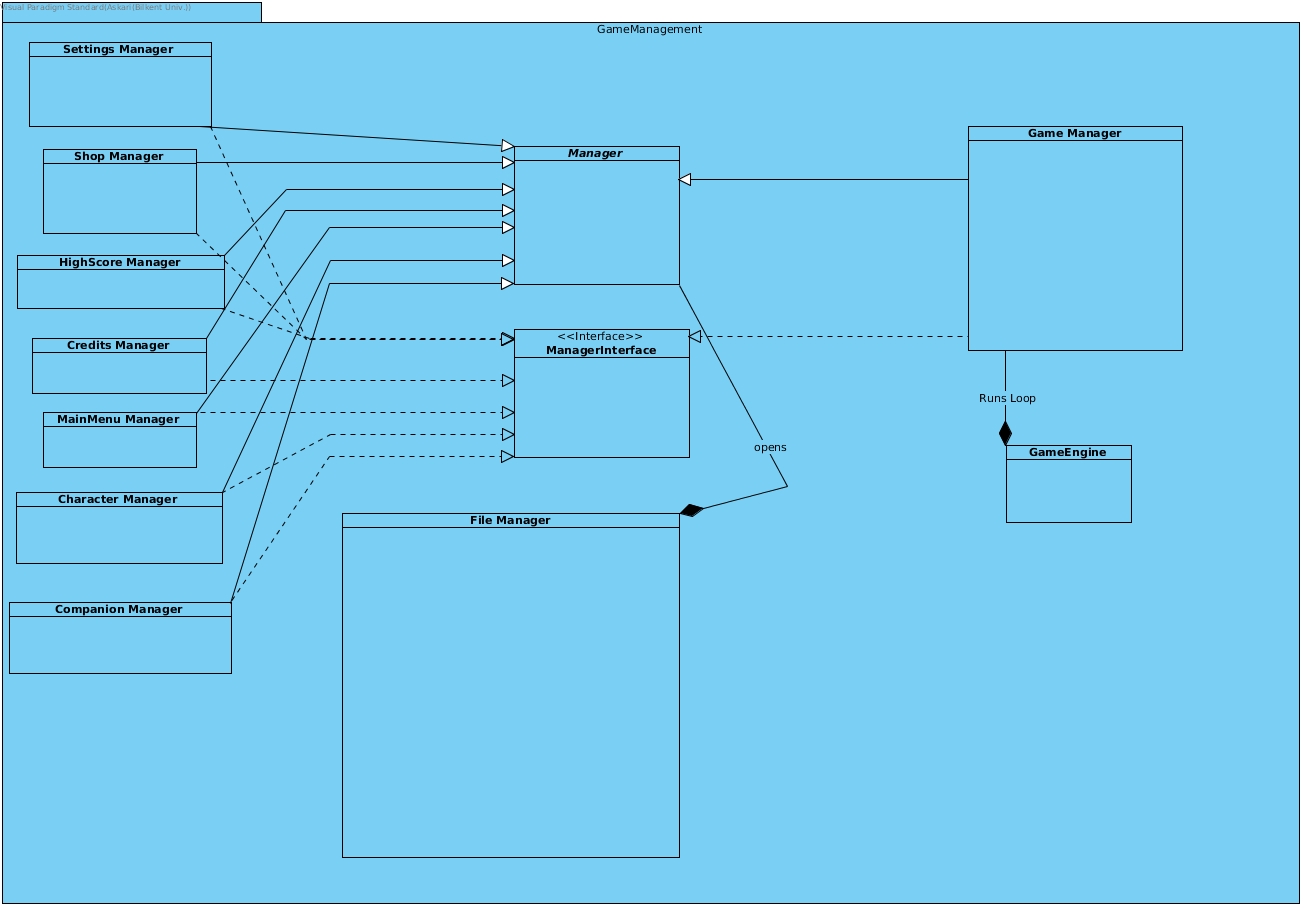
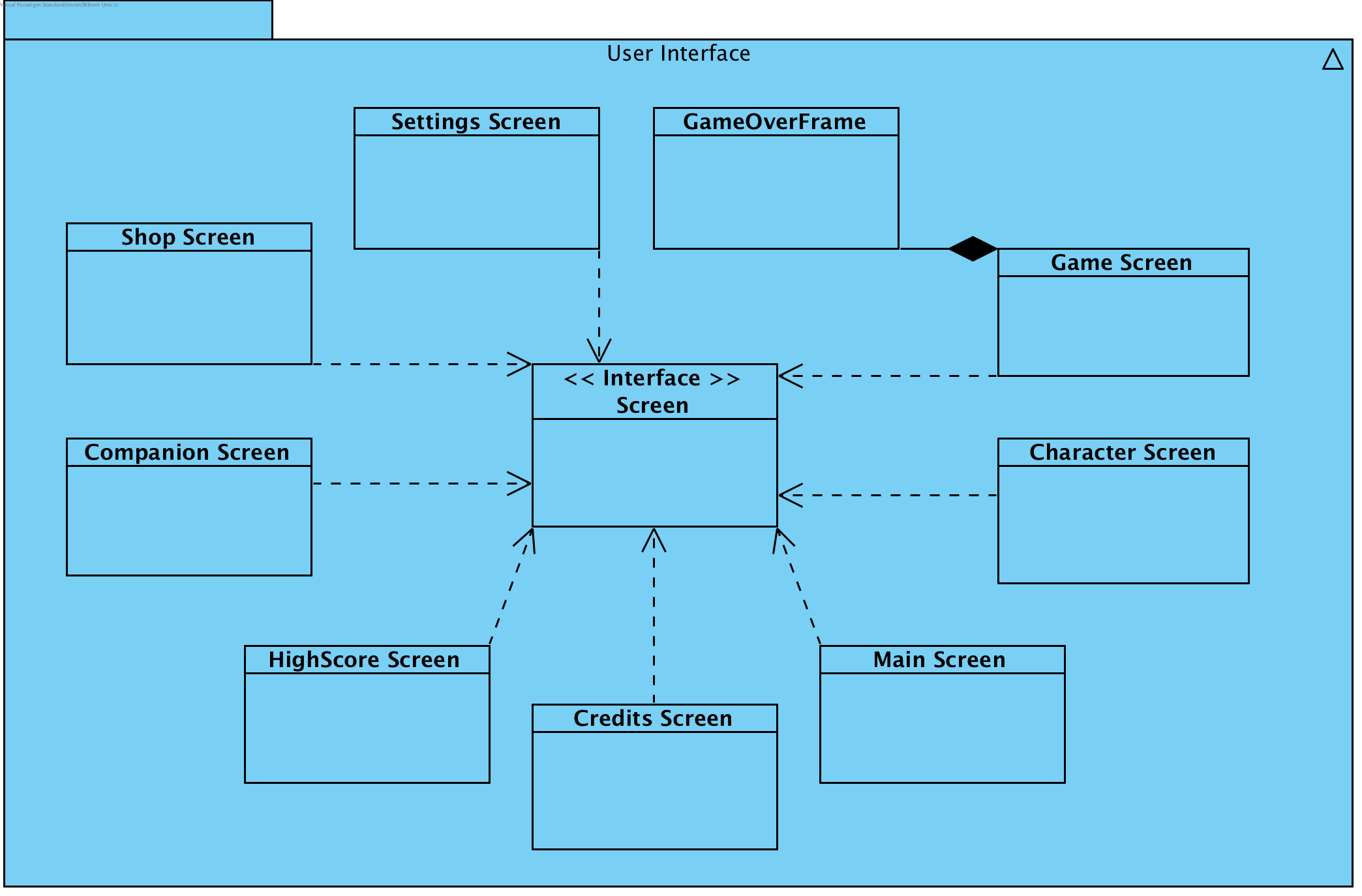
* + 1. **Definitions, Acronyms, Abbreviations**

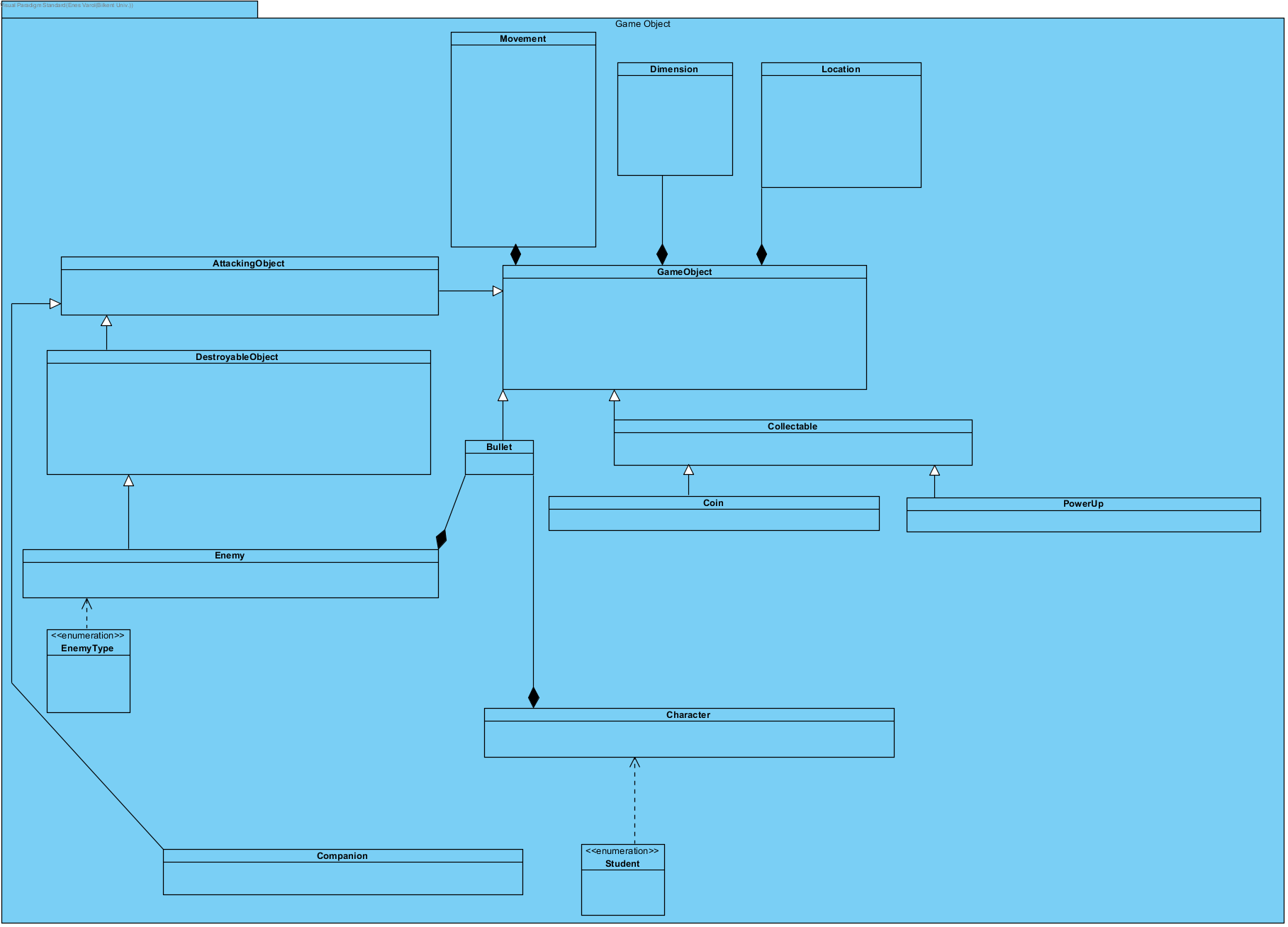
Java Virtual Machine - JVM

Model View Controller - MVC

Graphic User Interface - GUI

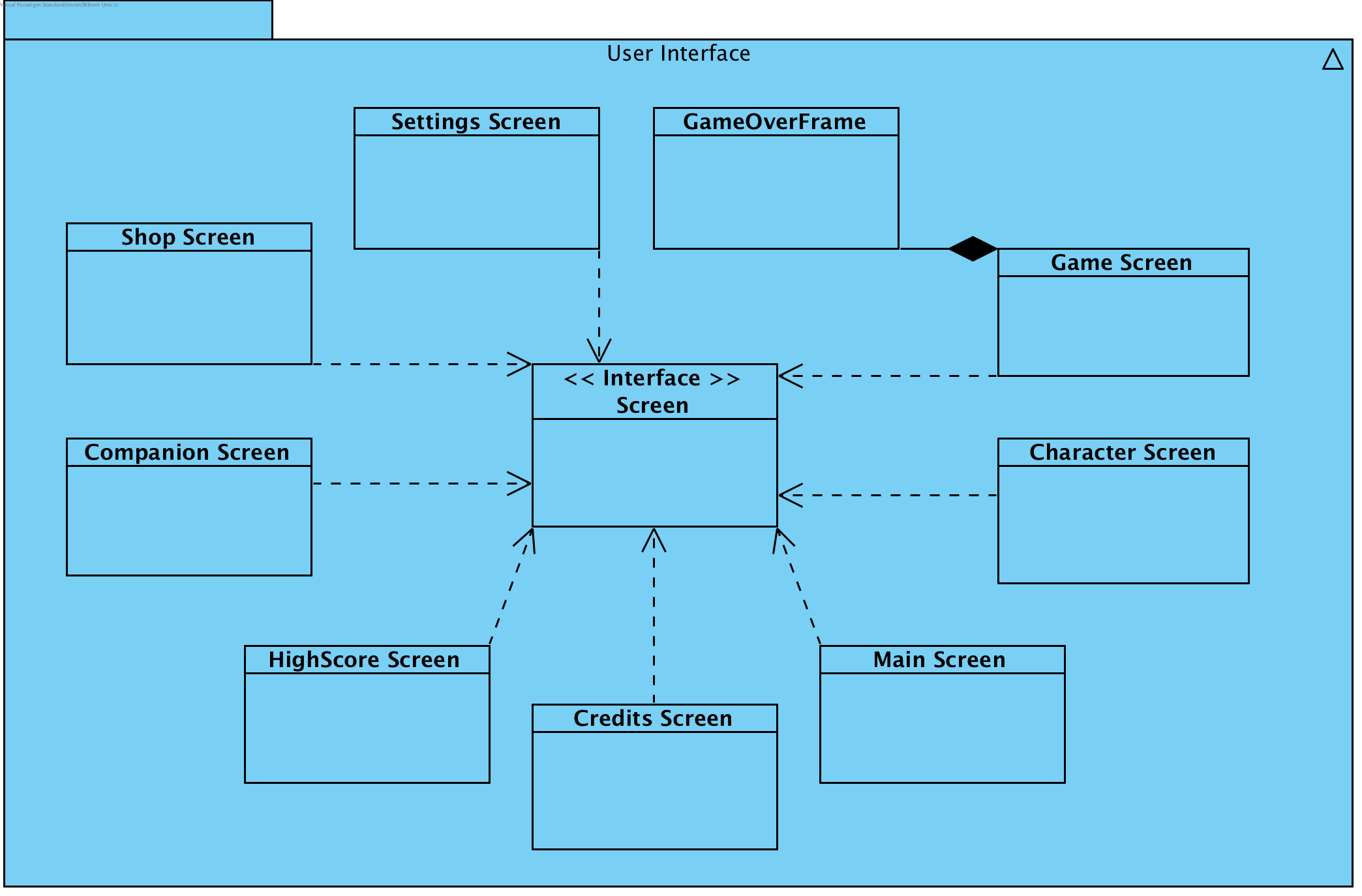
1. **System Architecture**
   1. **Subsystem Decomposition**



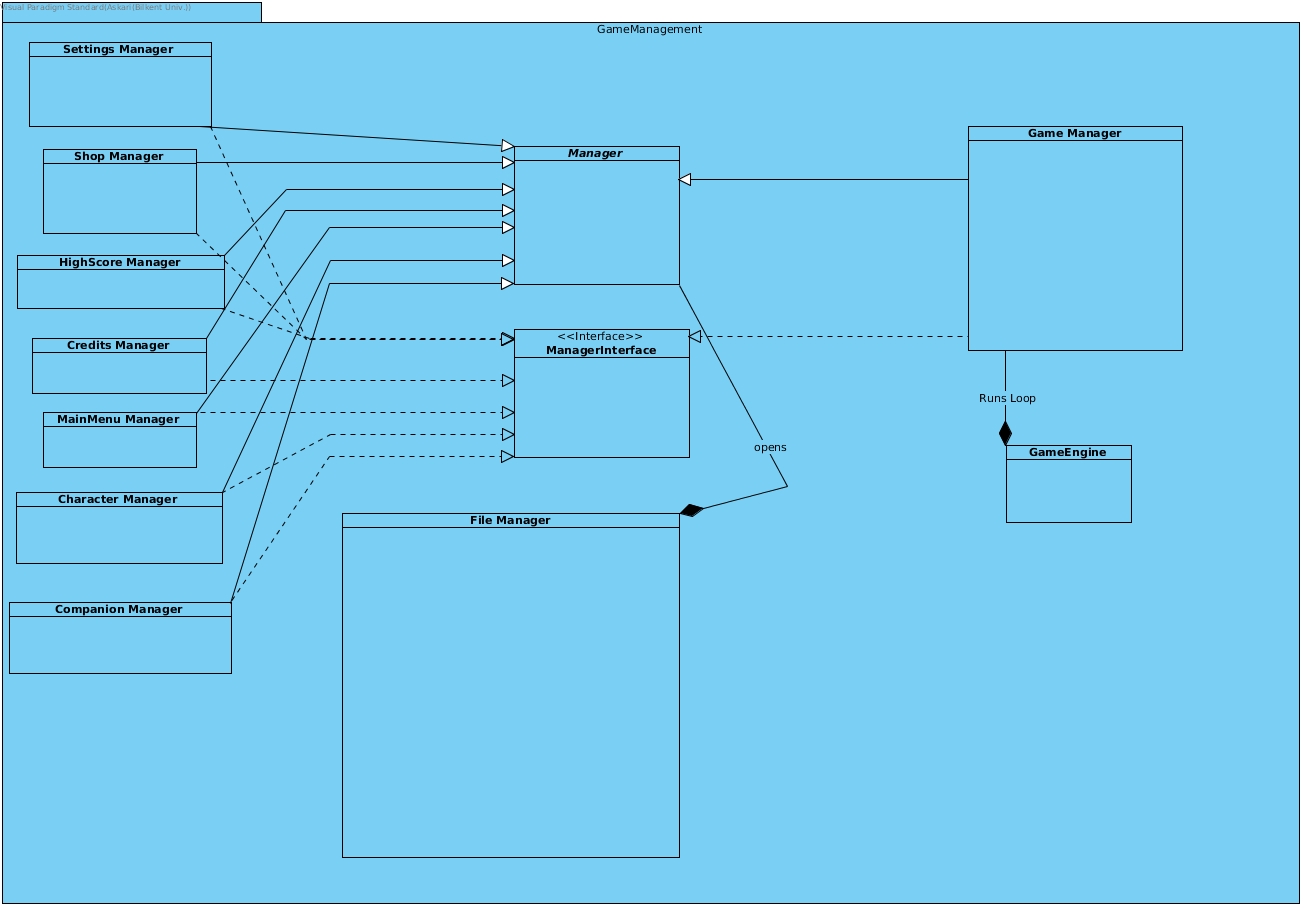


**Basic Subsystem Decomposition**

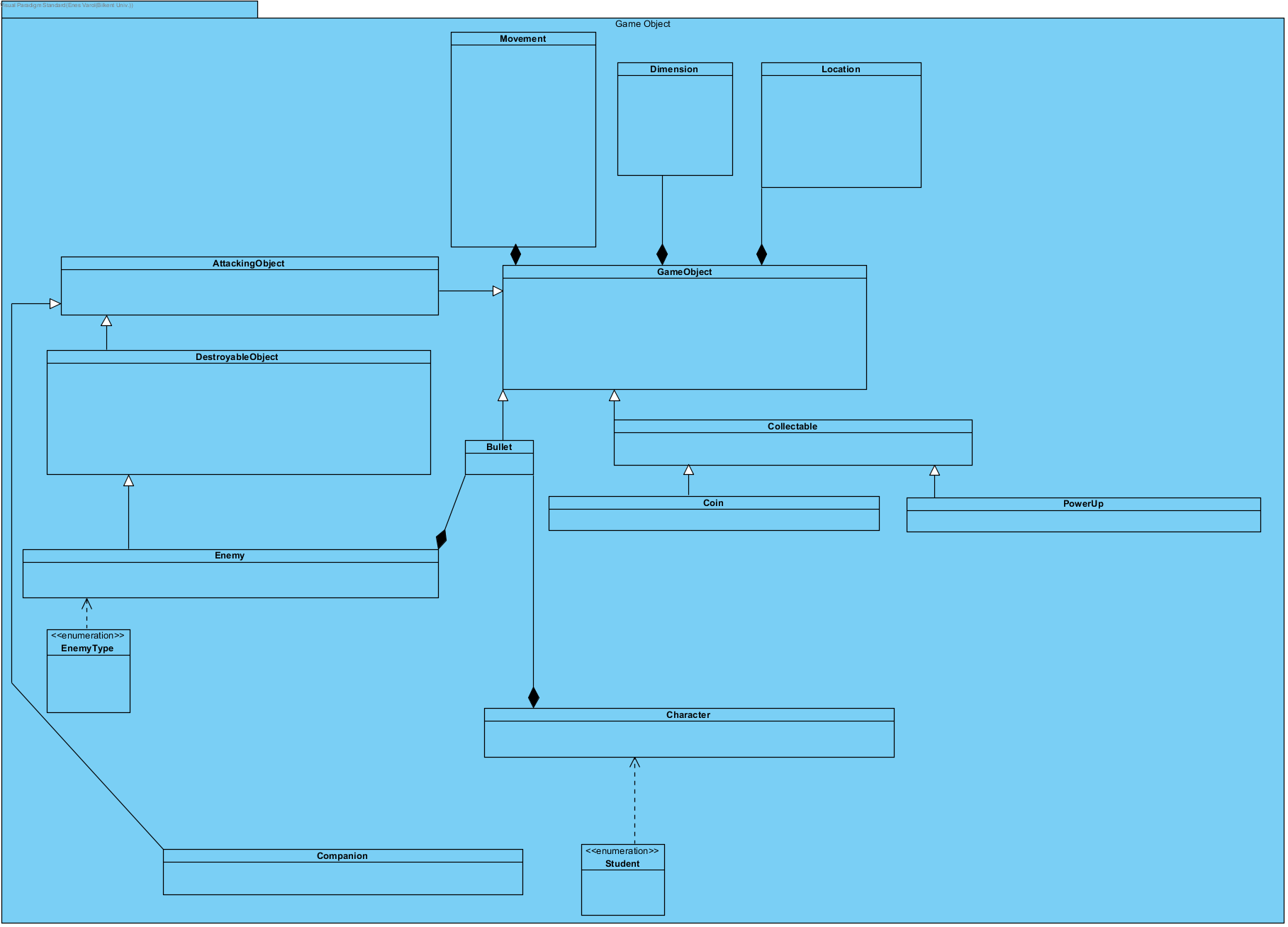
**Detailed Subsystem Decomposition**



**User Interface Subsystem**



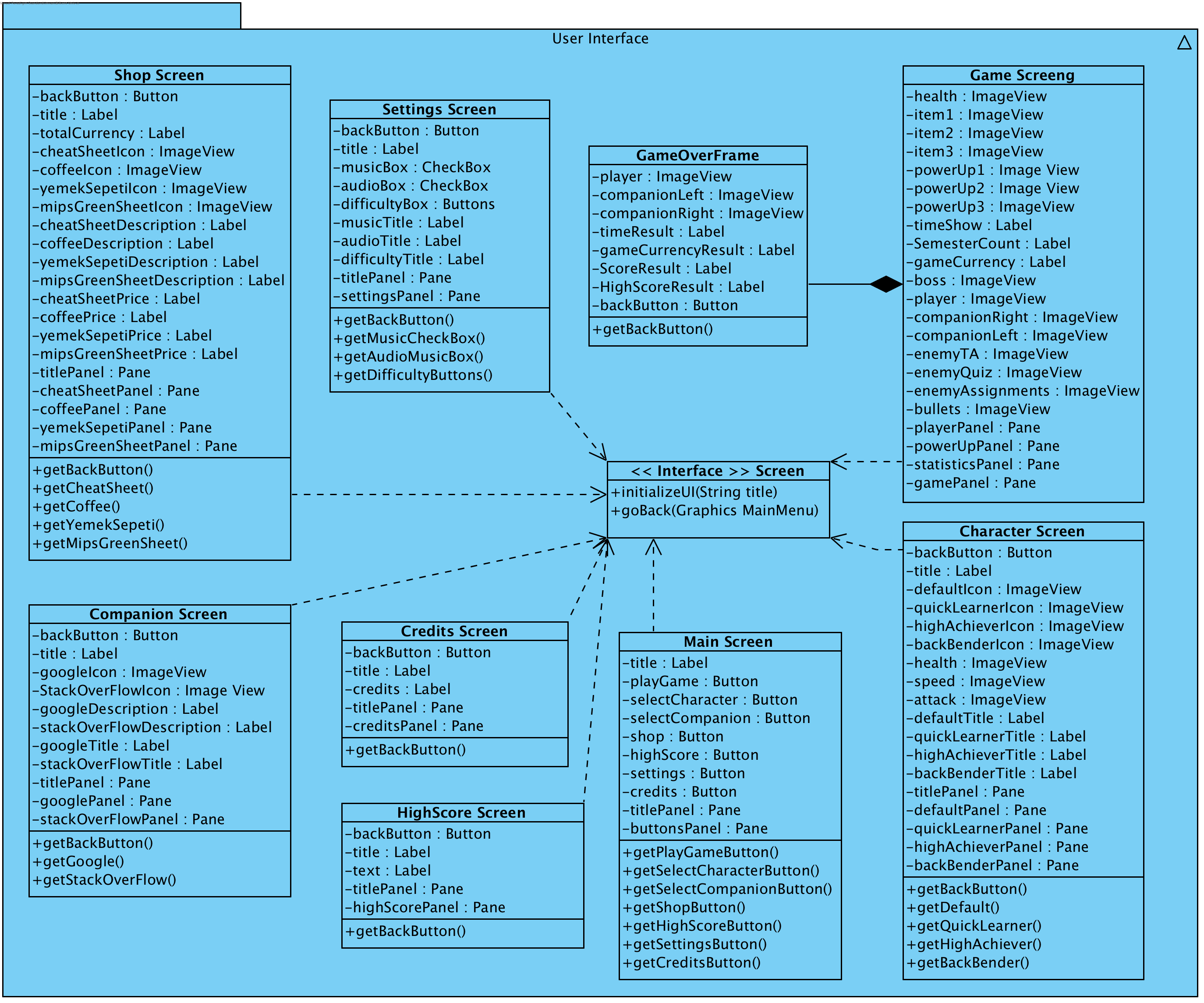
**Game Management Subsystem**



**Game Objects Subsystem**

1. **Subsystem Services**

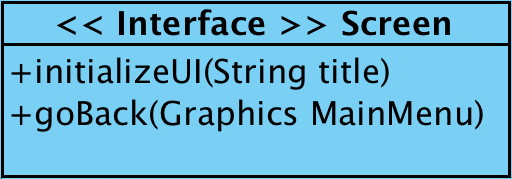
**3.1 User Interface Subsystem**



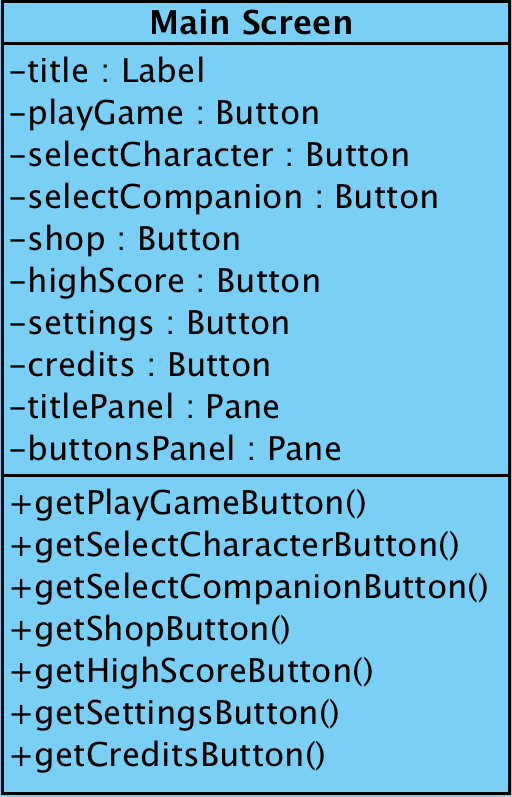
User Interface Subsystem of «A day in Bilkent» is composed of 8 different classes, each for one screen and a Frame that is going to be accessed in one of those 8 classes. All the classes extend the Screen Interface which consists of two methods: initializing the GUI and going back to the Main Menu. Main 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.

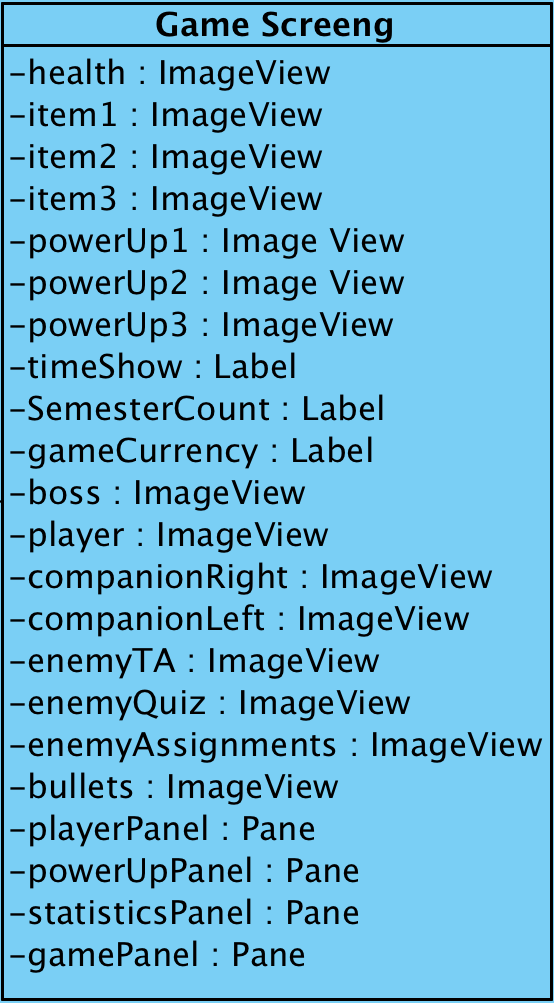
* public void goBack(Graphics MainMenu) - When user returns to Main Menu, loads it.

**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

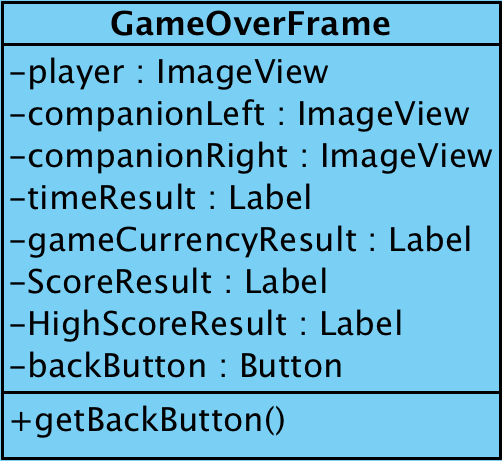
**Methods:**

* 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

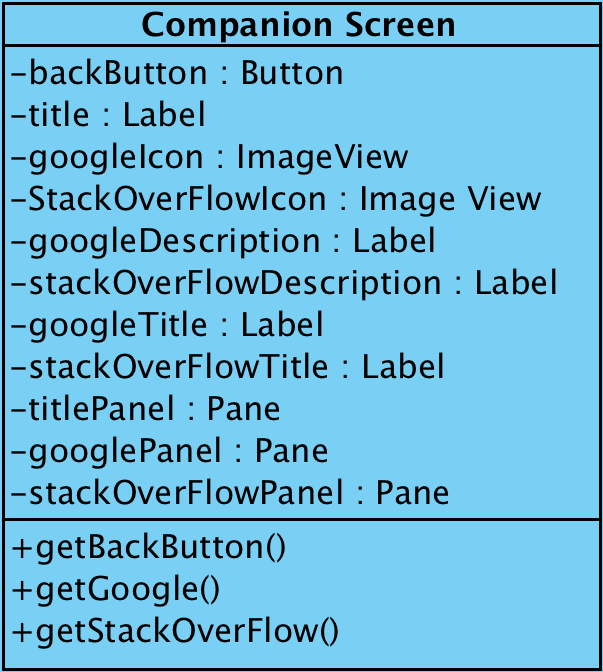
**Character Screen**

**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:**

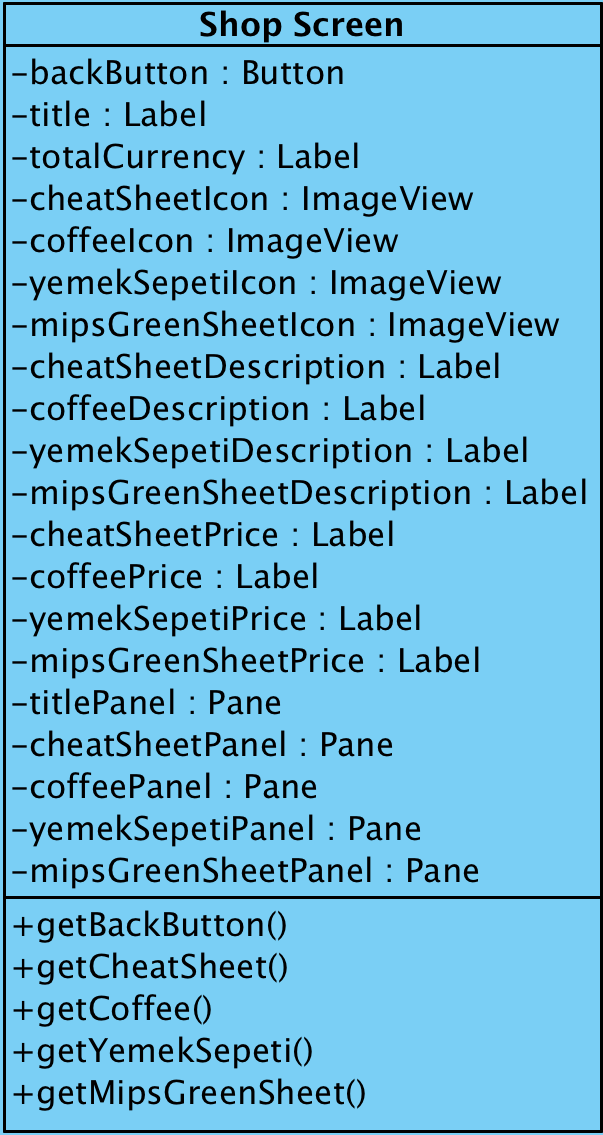
* public Button getBackButton() - returns the Back button to the controller
* public Button getDefault() - returns the Default character choice to the controller
* public Button getQuickLearner() - returns the Quick Learner character choice to the controller
* public Button getHighAchiever() - returns 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:**

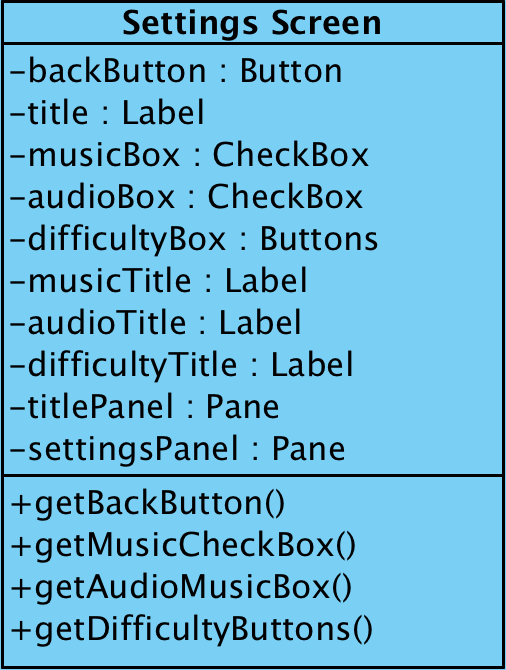
* public Button getBackButton() - returns the Back button to the controller
* public Button getGoogle() - returns the Google companion choice to the controller
* public Button getStackOverFlow() - returns the Stack OverFlow companion choice to the controller
* **Shop Screen**

**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

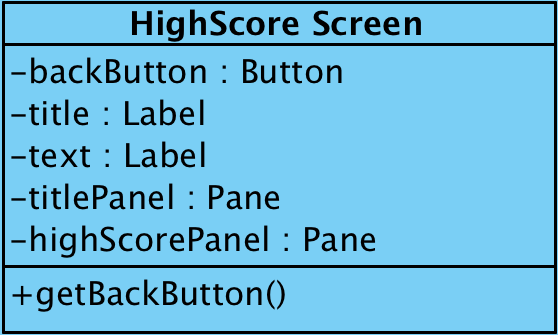
**Settings Screen**

**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

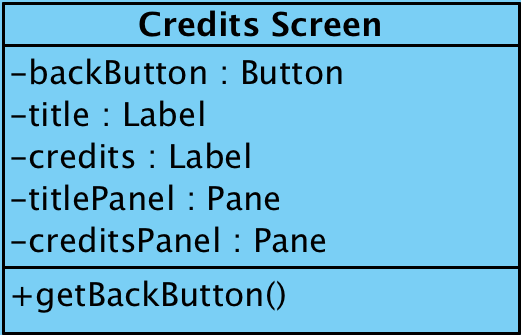
**HighScore Screen**

**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

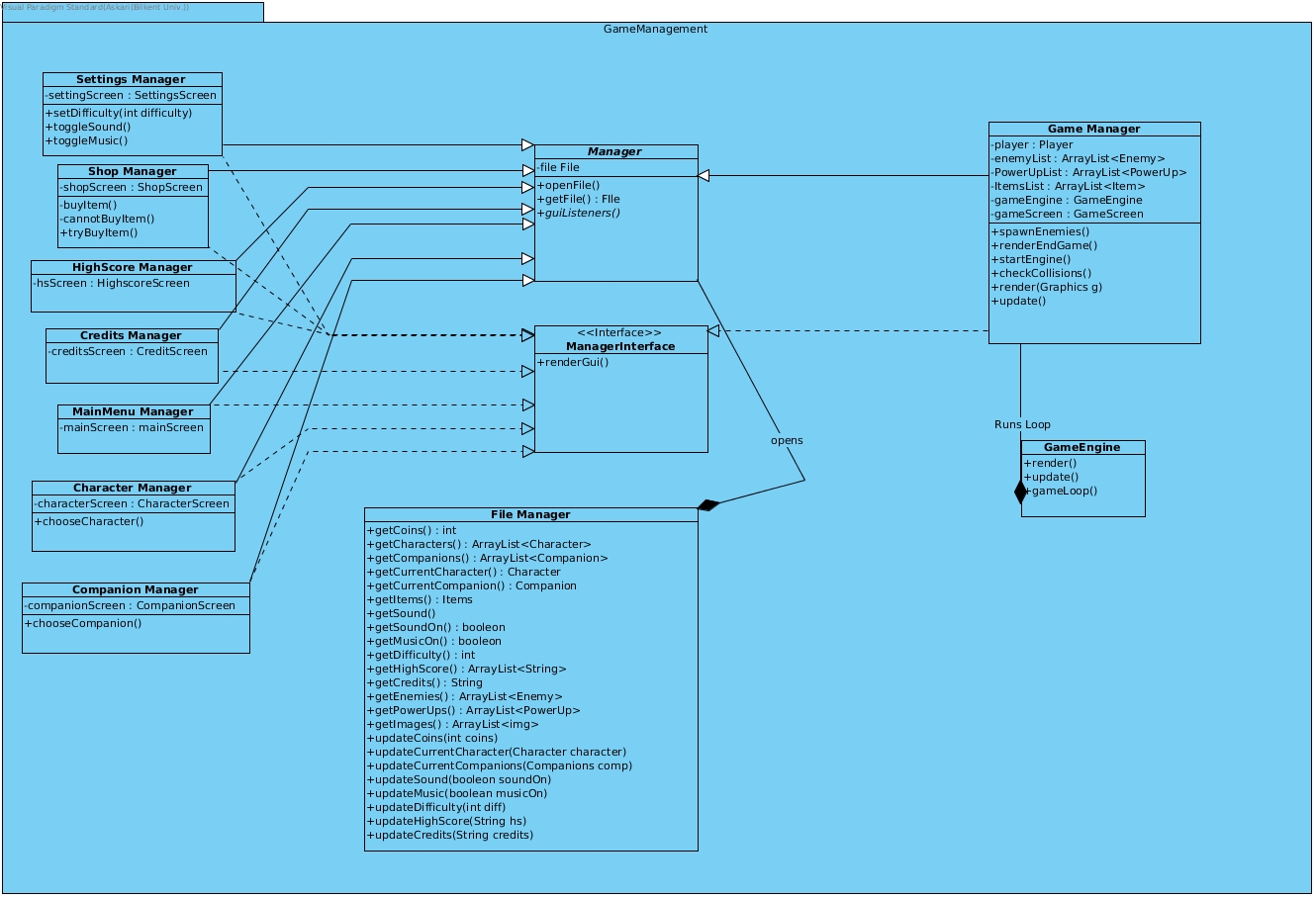
**Credits Screen**

**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**

**Figure 3.2 Detailed User Interface Subsystem**

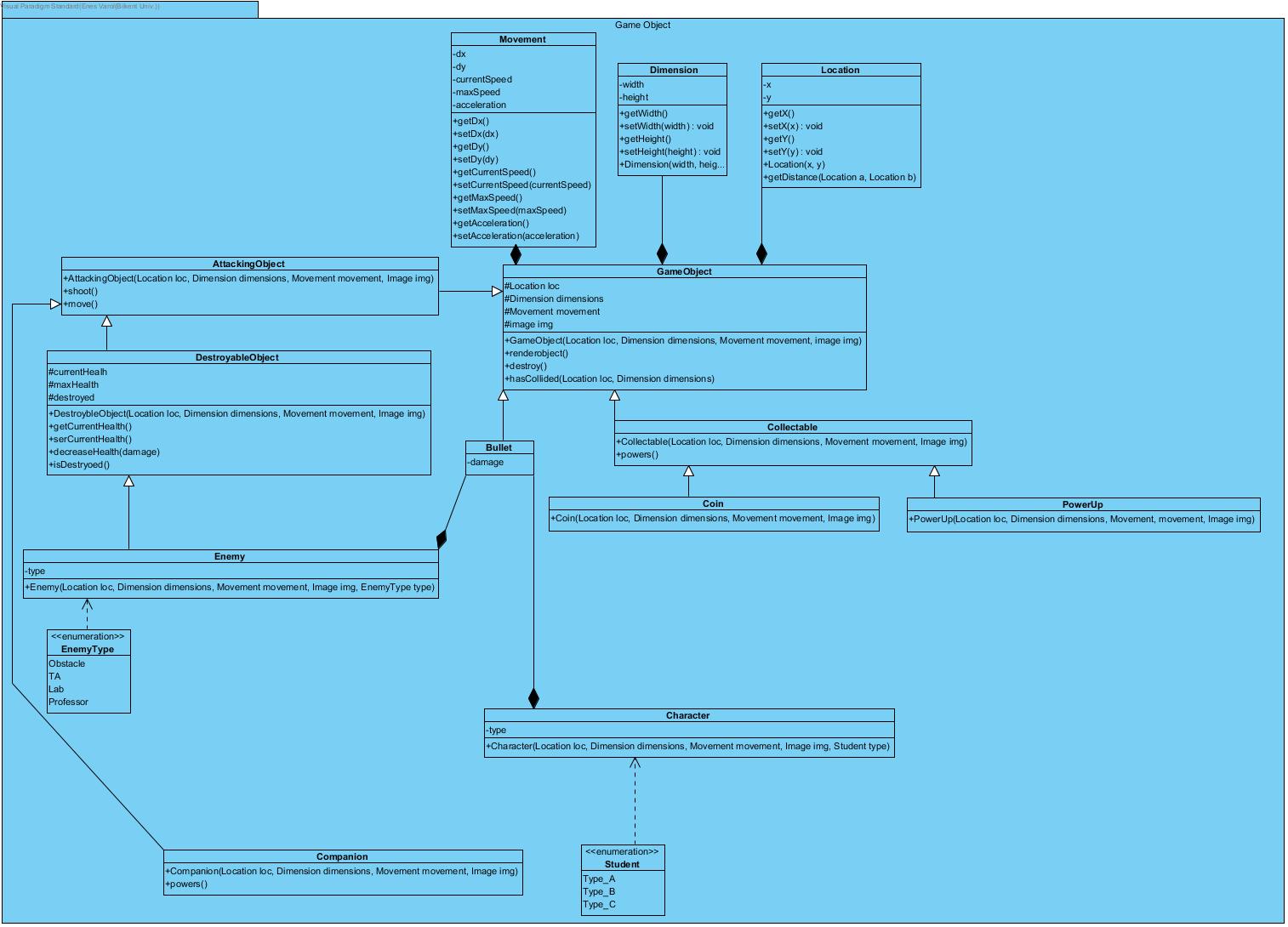
Game Management Subsystem of «A day in Bilkent» contains an interface and one master class called Manager that handles all the common functions of all managers. The managers are controllers for all the screens that are available and have to collect data, listen to button calls and send data back to the 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.

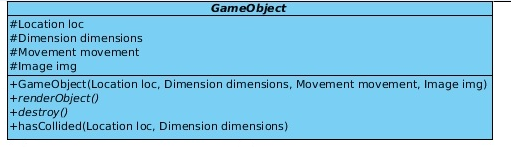
**3.3 Game Objects Subsystem**



Game Object Subsystem has 12 classes and shows the objects on the screen. The objects of the game are player(student), companions, enemies (Obstacle, Ta, Lab, Professor), Collectables (Coin, Power Up) and bullets. Subsystem holds those objects in classes: Collectable-Coin and Power Up, bullet, Destroyable Object- Enemy, Companion and Character.

Every object has their own methods to check their status but showing them on the screen is Game Object class’s job, this class shows Objects on the class with its renderobject() method.

**GameObject Class**



**Attributes:**

* private Location loc:Location information of the object
* private Dimension dimensions: Dimension information of the object
* private Movement movement: Movement information of the object
* private Image img: Object’s image

**Methods:**

* GameObject (Location loc, Dimension dimensions, Movement movement, Image img): Creates game objects
* renderObject (): renders Objects
* hasCollided (Location loc, Dimension dimensions): checks Objects collided or not

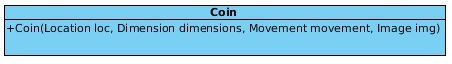
**Collectable Class**

Collectable

**Methods:**

* Collectable (Location loc, Dimension dimensions, Movement movement, Image img): Constructs collectable objects
* powers (): Activates the powers

**Coin Class**



**Methods:**

* Coin (Location loc, Dimension dimensions, Movement movement, Image img):Constructs coins

**PowerUp Class**

powerup

**Methods:**

* PowerUp (Location loc, Dimension dimensions, Movement movement, Image img): Constructs PowerUps

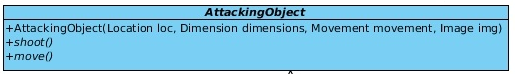
**Bullet Class**



**Attributes:**

* Damage:Attack value of bullet

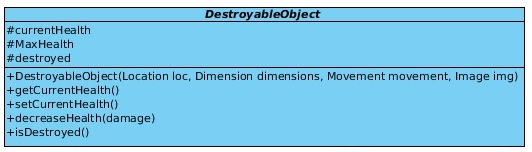
**AttackingObject class**



**Methods:**

* AttackingObject (Location loc, Dimension dimensions, Movement movement, Image img): Constructs attacking objects
* shoot (): Actiavtes shooting
* move (): Moves the objects

DestroyableObject class



**Attributes:**

* currentHealth: Holds health information
* Maxhealth: Holds object’s max health
* Destroyed: Boolean for destroyed information

**Methods:**

* DestroyableObject (Location loc, Dimension dimensions, Movement movement, Image img): Constructs destroyable objects
* getCurrentHealth (): returns current health
* setCurrentHealth (): sets current health
* decreaseHealth(damage): Decreases the health
* isDestroyed (): Checks whether destroyed or not

**Enemy Class**

enemy

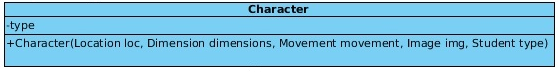
**Attributes:**

* type: Type information

**Methods:**

* Enemy (Location loc, Dimension dimensions, Movement movement, Image img, Enemytype type): Constructs enemy object

**Character Class**



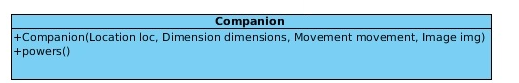
**Attributes:**

* type: Type information

**Methods:**

* Character (Location loc, Dimension dimensions, Movement movement, Image img, Student type): Constructs character objects

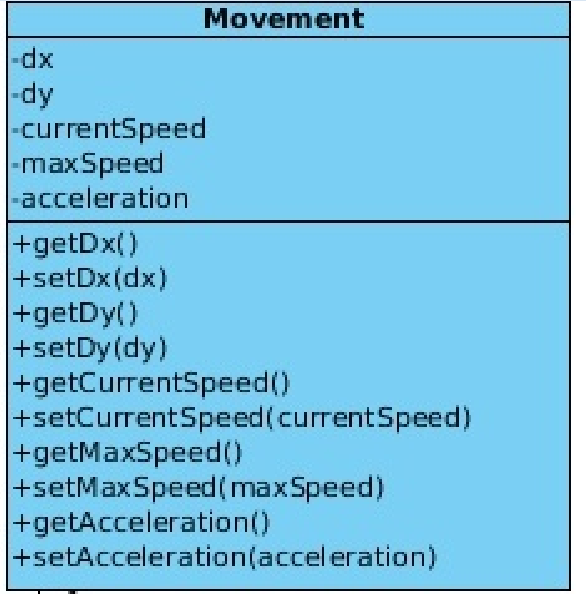
**Companion Class**



**Methods:**

* Companion (Location loc, Dimension dimensions, Movement movement, Image img):Constructs companion object
* powers (): Activates power

**Movement Class**



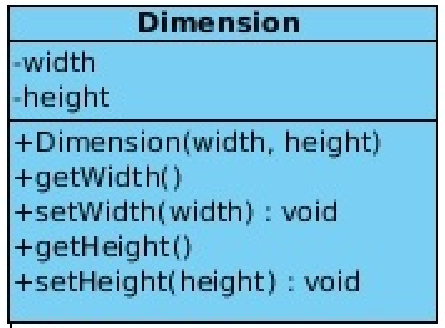
**Attributes:**

* dx: x coordinate information
* dy: y coordinate information
* currentSpeed: current speed information
* maxSpeed: maximum speed information
* acceleration: acceleration information

**Methods:**

* getDx():returns x
* setDx():sets x
* getDy():returns y
* setDy():set y
* getCurrentSpeed():returns current speed
* setCurrentSpeed(currentSpeed):sets current speed
* getMaxSpeed():returns maximum speed
* setMaxSpeed(maxSpeed):sets maximum speed
* getAccelaration():returns acceleraiton
* setAcceleration(acceleration):sets acceleration

**Dimension Class**



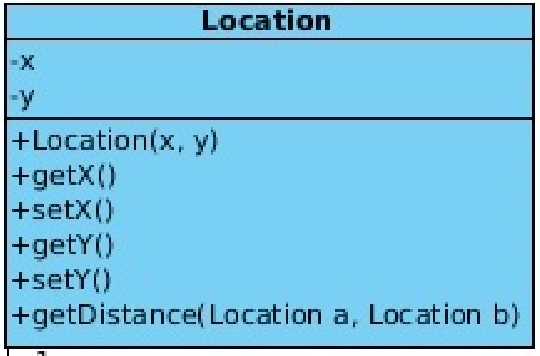
**Attributes:**

* width:width information
* height:height informaiton

**Methods:**

* Dimension(width, height):Constructs dimension for object
* getWidth():returns width
* setWidth(width):sets width
* getHeight():returns heeight
* setHeight(height):sets height

**Location Class**



**Attributes:**

* x: x coordinate
* y: y coordinate

**Methods:**

* Location(x,y):Constructs location object
* getX():returns x
* setX(x):sets x
* getY():returns y
* setY(x):sets y
* getDistance(Location a, Location b):returns the distance between 2 object

﻿**4. Low-level Design**

* 1. **Object design Trade-Offs**

**4.1.1 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. **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. **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. **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. **Packages**

**4.2.1 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.

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

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

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

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

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

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

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

* 1. **Class Interfaces**

**4.3.1 Screen**

This interface will be invoked whenever a UI initialization needed.

**4.3.2 Manager Interface**

This interface will be invoked when play game button pressed.

**5. References**

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