  
  
  
  
CS 319 - Object-Oriented Software Engineering  
Final Report  
  
  
Survivor  
Group 7

Defne Demirtürk

Sevde Şahin

Nihat Atay

Yağızhan Güler

Table of Contents

[1. Introduction 4](#_Toc437260203)

[2. Requirement Analysis 5](#_Toc437260204)

[2.1 Overview 5](#_Toc437260205)

[2.1.1 List of Bonuses 5](#_Toc437260206)

[2.1.2 List of Penalties 6](#_Toc437260207)

[2.1.3 List of Baits 7](#_Toc437260208)

[2.1.4 List of Cursors 8](#_Toc437260209)

[2.1.5 List of Other Objects 9](#_Toc437260210)

[2.2 Functional Requirements 9](#_Toc437260211)

[2.3 Non-functional Requirements 10](#_Toc437260212)

[2.4 Constraints 10](#_Toc437260213)

[2.5 Scenarios 11](#_Toc437260214)

[2.5.1. View High Scores 11](#_Toc437260215)

[2.5.3. Play Game 13](#_Toc437260216)

[2.5.4. Change Settings 14](#_Toc437260217)

[2.5.5. View Help 16](#_Toc437260218)

[2.5.6 View Credits 17](#_Toc437260219)

[2.6 Use-Case Models 18](#_Toc437260220)

[2.7 User Interface 19](#_Toc437260221)

[2.7.1 Navigational Path 23](#_Toc437260222)

[3. Analysis 24](#_Toc437260223)

[3.1 Object Model 24](#_Toc437260224)

[3.1.1 Domain Lexicon 24](#_Toc437260225)

[3.1.2 Class Diagrams 24](#_Toc437260226)

[3.2 Dynamic Models 26](#_Toc437260227)

[3.2.1 Activity Diagram 27](#_Toc437260228)

[3.2.2 Sequence Diagrams 28](#_Toc437260229)

[4. Design 34](#_Toc437260230)

[4.1 Design Goals 34](#_Toc437260231)

[4.2 Sub-System Decomposition 36](#_Toc437260232)

[4.3 Architectural Patterns 42](#_Toc437260233)

[4.3.1 Layers 42](#_Toc437260234)

[4.3.2 Model View Controller 43](#_Toc437260235)

[4.4 Hardware/Software Mapping 43](#_Toc437260236)

[4.5 Addressing Key Concerns 44](#_Toc437260237)

[4.5.1 Persistent Data Management 44](#_Toc437260238)

[4.5.2 Access Control and Security 44](#_Toc437260239)

[4.5.3 Global Software Control 45](#_Toc437260240)

[4.5.4 Boundary Conditions 45](#_Toc437260241)

[5. Object Design 46](#_Toc437260242)

[5.1 Pattern applications 46](#_Toc437260243)

[5.2 Class Interfaces 48](#_Toc437260244)

[5.3 Specifying Contracts 74](#_Toc437260245)

[6. Conclusions and Lessons Learned 75](#_Toc437260246)

**Table of Figures**

Figure 1: Accelerate Cursor

Figure 2: Extra Life Span

Figure 3: Expand Cursor

Figure 4: Rock

Figure 5: Bomb

Figure 6: Yellow Banana

Figure 7: Green Banana

Figure 8: Blue Banana

Figure 9: Red Banana

Figure 10: Regular Cursor

Figure 11: Red Cursor

Figure 12: Blue Cursor

Figure 13: Life Span

Figure 14: Use Case Diagram

Figure 15: Main menu window

Figure 16: In game window

Figure 17: Settings window

Figure 18: Help window

Figure 19: Credits window

Figure 20: Navigational Path

Figure 21: Class Diagram

Figure 22: Activity Diagram

Figure 23: Sequence Diagram (Start Game)

Figure 24: Sequence Diagram (Play Game)

Figure 25: Sequence Diagram (Change Settings)

Figure 26: Basic System Decomposition

Figure 27: Layers of System

Figure 28: Layer 1 and Layer 2

Figure 29: Layer 2 and Layer 3

Figure 30: User Interface Management Subsystem

Figure 31: Game Logic Subsystem

Figure 32: Input Management Subsystem

Figure 33: Game Elements Subsystem

Figure 34: File Management Subsystem

Figure 35: User Interface Management Subsystem

Figure 36: Main Menu Class

Figure 37: Settings Panel Class

Figure 38: ViewPanel Class

Figure 39: Pause Menu Class

Figure 40: Game Logic Subsystem

Figure 41: Game Engine Class

Figure 42: Graphics Visualizer Class

Figure 43: Sound Manager Class

Figure 44: Game Elements Subsystem

Figure 45: Game Object Class

Figure 46: Penalty Class

Figure 47: Rock Class

Figure 48: Bomb Class

Figure 49: Ray Class

Figure 50: Bonus Class

Figure 51: Life Span Class

Figure 52: Extra Point Class

Figure 53: Extra Speed Class

Figure 54: Bait Class

Figure 55: Red Banana Class

Figure 56: Yellow Banana Class

Figure 57: Blue Banana Class

Figure 58: Green Banana Class

Figure 59: Cursor Class

Figure 60: Type of cursor object

Figure 61: Input Manager Subsystem

Figure 62: Input Manager Class

Figure 63: Input Manager Class

Figure 64: File Manager Class

# 1. Introduction

Survivor is an action game basically evolved from DXBall that we decided to develop. We plan to develop our game with different features and images that we have seen from DXBall.

In Survivor, we are going to use new features as well. In the game, the aim of the user is to catch baits fall from top. By this, the user will be able to increase points and so level up. However, the baits that fall out will include rocks, which will decrease the player’s points, and bombs which will cause the player to lose 1 life span. Also the baits fall out will include extra life span, which will give extra playing chance to user, and extra speed, which will give extra speed to cursor for a few seconds.

Apart from these, there will be enemy against user. This immobile machine will be sending ray to curser and user will try to escape. If this ray hits curser, user will lose 1 life span to play game. Furthermore, the game will be a desktop application and will be controlled by a mouse or the keyboard.

# 2. Requirement Analysis

## 2.1 Overview

The game consists of five levels where the colors of baits are changed. For each level there will be 10000 points limit. Purpose of this game is completing all the levels without losing all lives. Basically, the game is about collecting falling baits by the cursor to achieve goal points. While the player playing the game there can be various bonuses as well as the rocks and bombs. The bonuses provide advantages to player while rocks make the player lose points. Moreover, collected bombs cause to lose life span.

During the game, player can pause and carry on game at any time.

There will be high score system which will keep the highest 10 scores with players’ name. Player gains according to the type of bananas. Different types of bananas give different amount of points.

Player will be able to control cursor using keyboard.

Player is allowed to change game background and cursor using the setting menu. There will be three different background and cursor options. Also user can change game sound as on and off.

### 2.1.1 List of Bonuses



Figure 1: Accelerate cursor

It accelerates the cursor.



Figure 2: Extra life span

It gives extra life span to user.



Figure 3: Expand cursor

It expands the cursor.

### 2.1.2 List of Penalties



Figure 4: Rock

It causes 5000 point loss.



Figure 5: Bomb

It causes 1 life span.

### 2.1.3 List of Baits

C:\Users\Nihat\Desktop\mockups\banana 1.png

Figure 6: Yellow Banana

It gives 250 points to player.

C:\Users\Nihat\Desktop\mockups\banana 2.png

Figure 7: Green Banana

It gives 500 points to player.

C:\Users\Nihat\Desktop\mockups\banana 3.png

Figure 8: Blue Banana

It gives 1000 points to player.

C:\Users\Nihat\Desktop\mockups\banana 4.png

Figure 9: Red Banana

It gives 2500 points to player.

### 2.1.4 List of Cursors

C:\Users\Nihat\Desktop\mockups\Cursor.png

Figure 10: Regular Cursor

This is the default cursor in game.

C:\Users\Nihat\Desktop\mockups\cursor 2.png

Figure 11: Red Cursor

Player can choose this cursor in settings menu.

C:\Users\Nihat\Desktop\mockups\cursor 3.png

Figure 12: Blue Cursor

Player can choose this cursor in settings menu.

### 2.1.5 List of Other Objects

C:\Users\Nihat\Desktop\mockups\Life Span.png

Figure13: Life Span

Next to this object, user can see how many life span s/he has.

## 2.2 Functional Requirements

* User will be able to control the cursor with mouse.
* User can change the settings of the game. Such as:

Change the background

Change the sounds

* User will be able to access the help menu that consists of information about how to play the game and items.
* User can see the high scores of top 10 players.
* The user shall be allowed to pause the game and then resume.

## 2.3 Non-functional Requirements

* Control mechanism of the game shall have short response time that allows the player to play with minimal delay.
* Concepts of the levels in the game are easy to understand and react to.
* The game shall offer numerous bonuses and penalties.
* The cursor shall respond at most 1 millisecond.

## 2.4 Constraints

* The game will be implemented in Java.
* The game will be played in 60 FPS.
* Draws in the game will be smooth.
* The game will run all operating systems that support Java.

## 2.5 Scenarios

### **2.5.1. View High Scores**

**Use Case Name**: View High Scores

Primary Actor: Player

**Entry Condition**: Player selects “View High Scores” option from Main Menu.

**Exit Condition**: Player selects “Back” option to return to Main Menu.

Event Flow:

-Player wants to see top ten scores with player names.

-System shows the list containing top ten scores with player names.

**Pre-conditions**: System keeps records of top ten scores.

Post-condition: -

Success Scenario Event Flow:

1. System displays top ten scores with player names.

Alternative Flows:

-If player desires to return to Main Menu at any time:

1) Player selects “Return to Main Menu” button to return to Main Menu.

2) System displays Main Menu.

**2.5.2. View Credits**

**Use Case Name**: View Credits

Primary Actor: Player

**Entry Condition**: Player selects “View Credits” from Main Menu.

**Exit Condition**: Player selects “Back” to return to Main Menu.

Event Flow:

- Player wants to learn names of people and their contact information created Survivor.

- System displays contact information of software developers with their names.

**Pre-conditions**: Player should be in the Main Menu window.

Post-condition: -

Success Scenario Event Flow:

1. System displays contact information and names of people who developed Survivor.

Alternative Flows:

- If player desires to return main menu at any time:

1) Player selects “Return to Main Menu” button to return main menu.

2) System displays Main Menu.

### **2.5.3. Play Game**

Use Case Name: Play Game

Primary Actor: Player

**Entry Condition**: Player selects “Play Game” button from Main Menu.

**Exit Condition**: Player selects “Return to Main Menu” from the game screen.

Event Flow:

-Player aims to complete all levels and make highest score.

-System keeps the score of the Player.

**Pre-condition**: For first running, game settings are set as default. If Player changes game settings, adjusted settings will be saved and used by System.

**Post-condition**: If Player gets a score which is high enough to be in high score list, high score list will be updated by System.

Success Scenario Event Flow:

1. Game is started by the System.

2. Player starts playing from first level.

3. Player plays the level until the time limit is reached.

4. System takes the player to the next level.

5. Player starts playing next level.

Player repeats the steps 3 – 5 until all levels are completed or Player loses all lives.

6. System record Player’s score to High Score List, if Player score is higher than the lowest score in high score list and displays the High Score List.

7. System returns to Main Menu.

Alternative Flows:

A) Player collects the bonuses during game:

1. Cursor collides with bonuses.

2. Player tries to collect bonuses by using his cursor.

3. System removes the bonus, whether user collects it or bonus falls from screen.

4. If user collects the bonus System makes necessary changes to apply the feature of bonus and system updates score of the Player according to bonus.

### **2.5.4. Change Settings**

**Use Case Name**: Change Settings

Primary Actor: Player

**Entry Condition**: Player selects “Change Settings” button from Main Menu.

**Exit Condition**: Player selects “Back” to return to Main Menu.

Event Flow:

-Player desires to change game settings: disabling game sounds or enabling game sound, changing the background, changing the cursor type.

-System updates the settings which are changed by player.

**Pre-condition**: For first running, game settings will be set as default. If Player changes game settings, adjusted settings will be saved and used by system.

**Post-condition**: Game settings are updated.

Success Scenario Event Flow:

1. Player presses “Change Settings” button to make changes about game settings.

2. Game settings are displayed to Player in “Change Settings” screen by System.

3. Player adjusts settings according to his desire.

4. System updates game settings successfully.

Alternative Flows:

A. If Player desires to use default settings at any time:

1. Player selects “Default Settings” button from “Change Settings” screen.

2. Default game settings are updated by System.

B. If Player requests to return previous menu at any time:

1. Player selects “Back” button from “Change Settings” screen.

2. Game settings are updated by System.

3. Player returns to Main Menu.

### **2.5.5. View Help**

Use Case Name: View Help

Primary Actor: Player

**Entry Condition**: Player selects “View Help” from Main Menu.

**Exit Condition**: Player selects “Back” to return to Main Menu.

Event Flow:

- Player wants to learn purpose of game and how to play it.

- System shows a text document explaining purpose of game, player controls, bonuses and penalty object types.

**Pre-conditions**: Player should be in Main Menu.

Post-condition: -

Success Scenario:

1. Player selects “View Help” from Main Menu.

2. System displays help document giving instructions about main purpose of game, game controls and bonuses, baits, bombs and rocks.

Alternative Flows:

A. If Player requests to return previous menu at any time:

1. Player selects “Back” button from “View Help” screen.

2. Player returns to Main Menu.

### 2.5.6 View Credits

**Use Case Name**: View Credits

Primary Actor: Player

**Entry Condition**: Player selects “View Credits” from Main Menu.

**Exit Condition**: Player selects “Back” to return to Main Menu.

Event Flow:

- Player wants to learn names of people and their contact information created Survivor.

- System displays contact information of software developers with their names.

**Pre-conditions**: Player should be in the Main Menu window.

Post-condition: -

Success Scenario Event Flow:

1. System displays contact information and names of people who developed Survivor.

Alternative Flows:

- If player desires to return to Main Menu at any time:

1) Player selects “Return to Main Menu” button to return to Main Menu.

2) System displays Main Menu.

## 2.6 Use-Case Models

This section provides information about the main use case model of Survivor game, detailed use case explanations are included below.

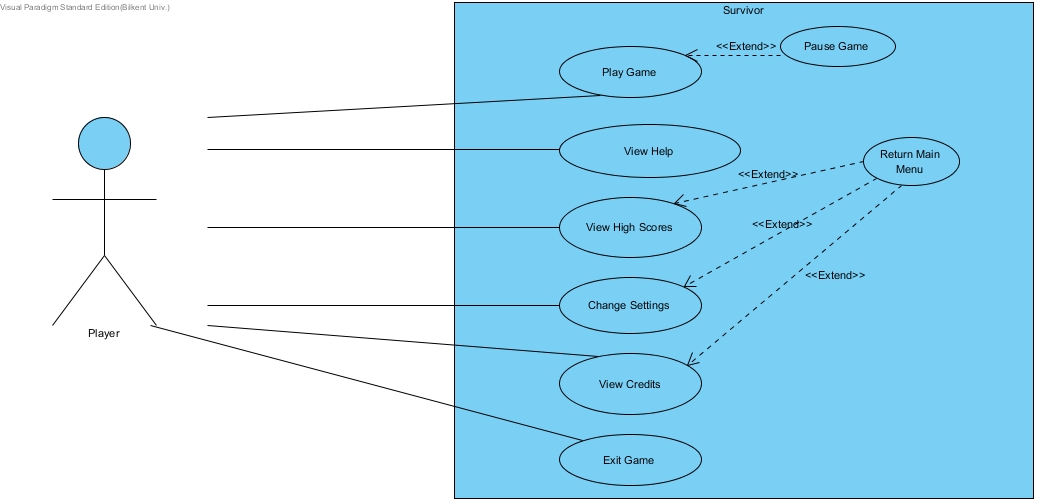


Figure 14: Illustrates the use case model of Survivor

## 2.7 User Interface

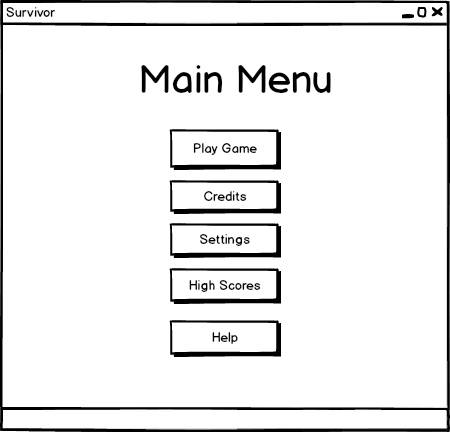


Figure 15: Main Menu Window



Figure 16: In game window

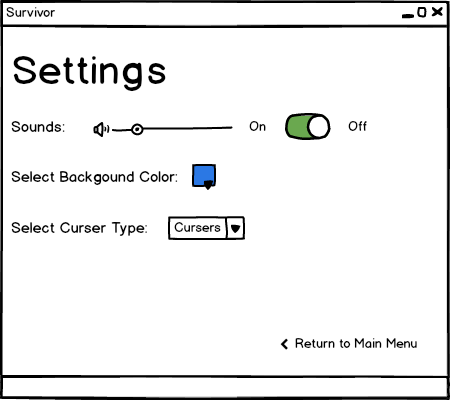


Figure 17: Settings window

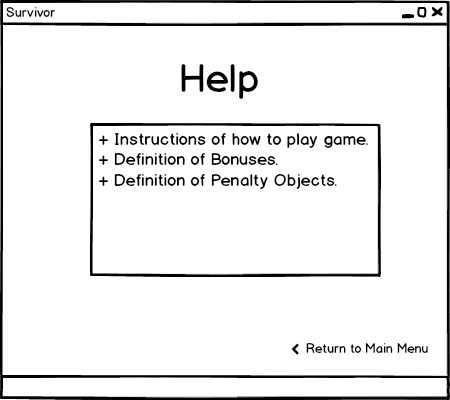


Figure 18: Help Window

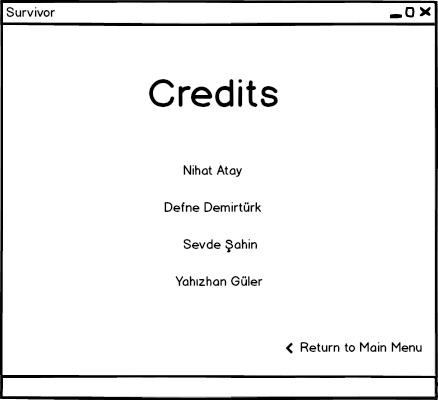
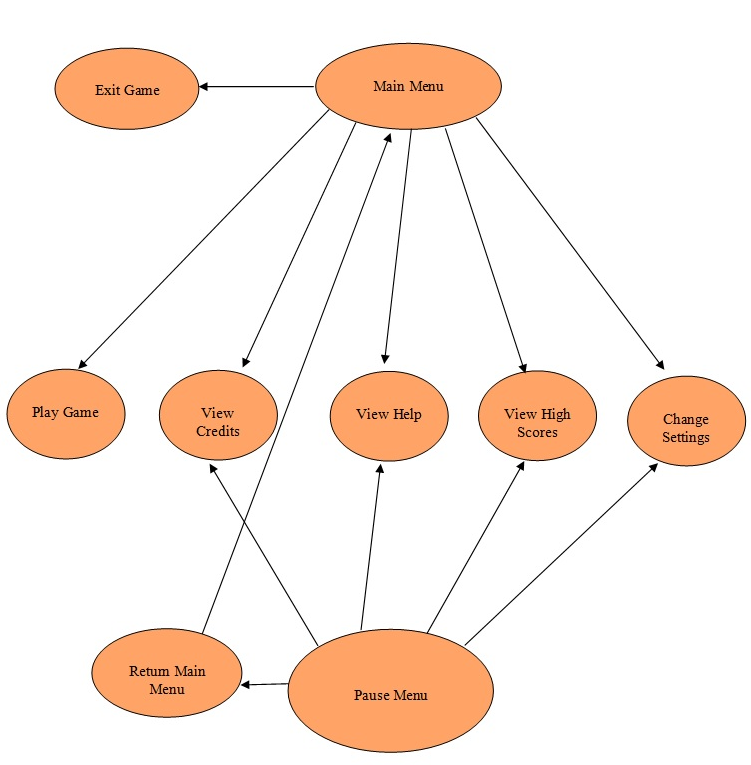


Figure 19: Credits Window

### 2.7.1 Navigational Path



### 

Figure 20: Navigational Path

# 3. Analysis

## 3.1 Object Model

### 3.1.1 Domain Lexicon

### 3.1.2 Class Diagrams

The game Survivor consists of 16 classes.

**GameEngine** class is the class where the main management of the game is handled. It manages the lifecycle of the game by cooperating with other manager classes. It stores the score, high score and time features of the game.

**CollisionManager** class reports the collision type that occurs in the current state of the game and also it updates game objects to the GameEngine when needed.

**InputManager** class carries the management of inputs which are basically the interruptions that are caused by buttons.

**MainMenu** class handles all options of the game by contacting with the related classes. It also represents the features of our user interface.

**FileManager** class handles the connection between the files associated with the game. It handles mostly recording the scores of players.

Furthermore, GameObject class consists of 5 different types of objects. Namely, Bait, Cursor, Ray, Bonus and Penalty classes. We have Penalty class and it has 2 child classes, named Rock and Bomb. These are the objects which have negative effect on the gameplay. Moreover, we have 3 different types of bonuses named, ExtraSpeed, ExtraPoint and LifeSpan. These are the objects which have positive effect on the gameplay.

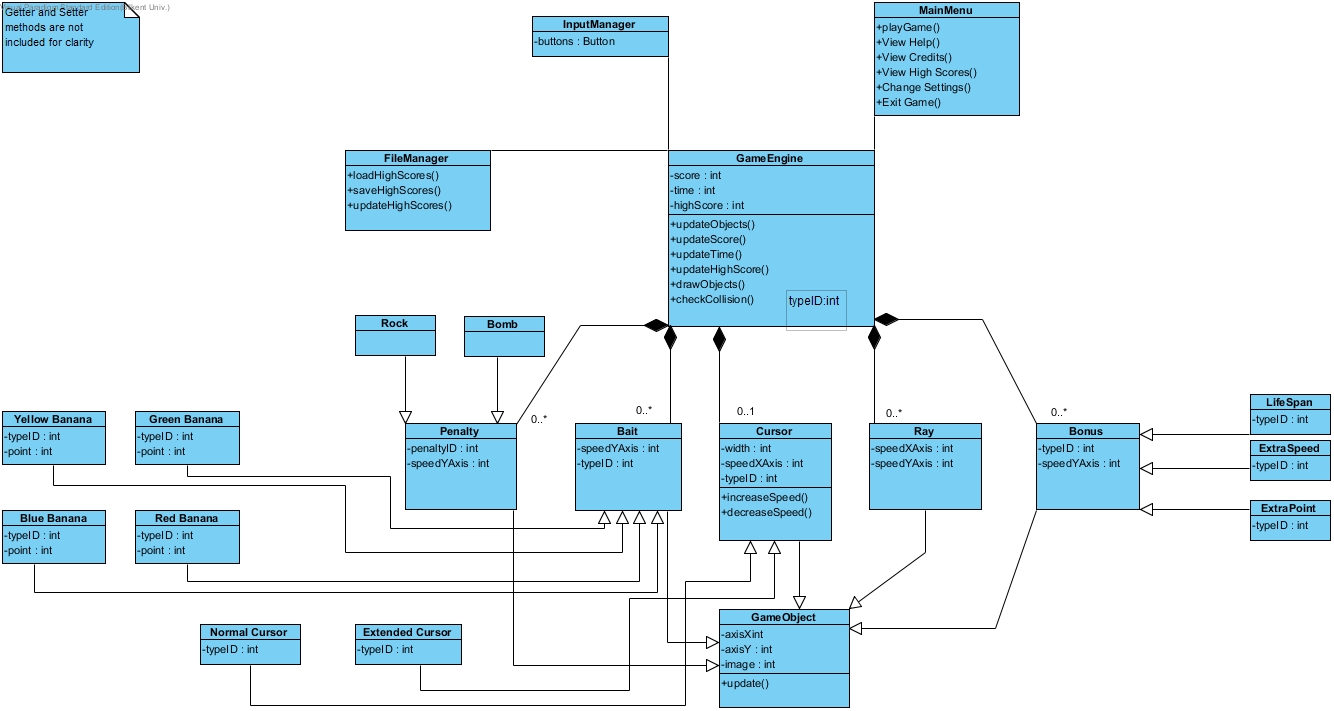


Figure 21: Class Diagram of Survivor Game

## 3.2 Dynamic Models

This part provides detailed information about Survivor game by showing some important scenarios of the game in sequence diagrams and state diagrams. Since the GameEngine, FileManager, Player and InputManager objects are the important objects of Survivor, behavioral scenarios of these particular objects are described in sequence diagrams. Moreover, the Bait, Cursor, Player and Ray objects are also important objects so their states are examined in the state chart diagrams.

### 3.2.1 Activity Diagram

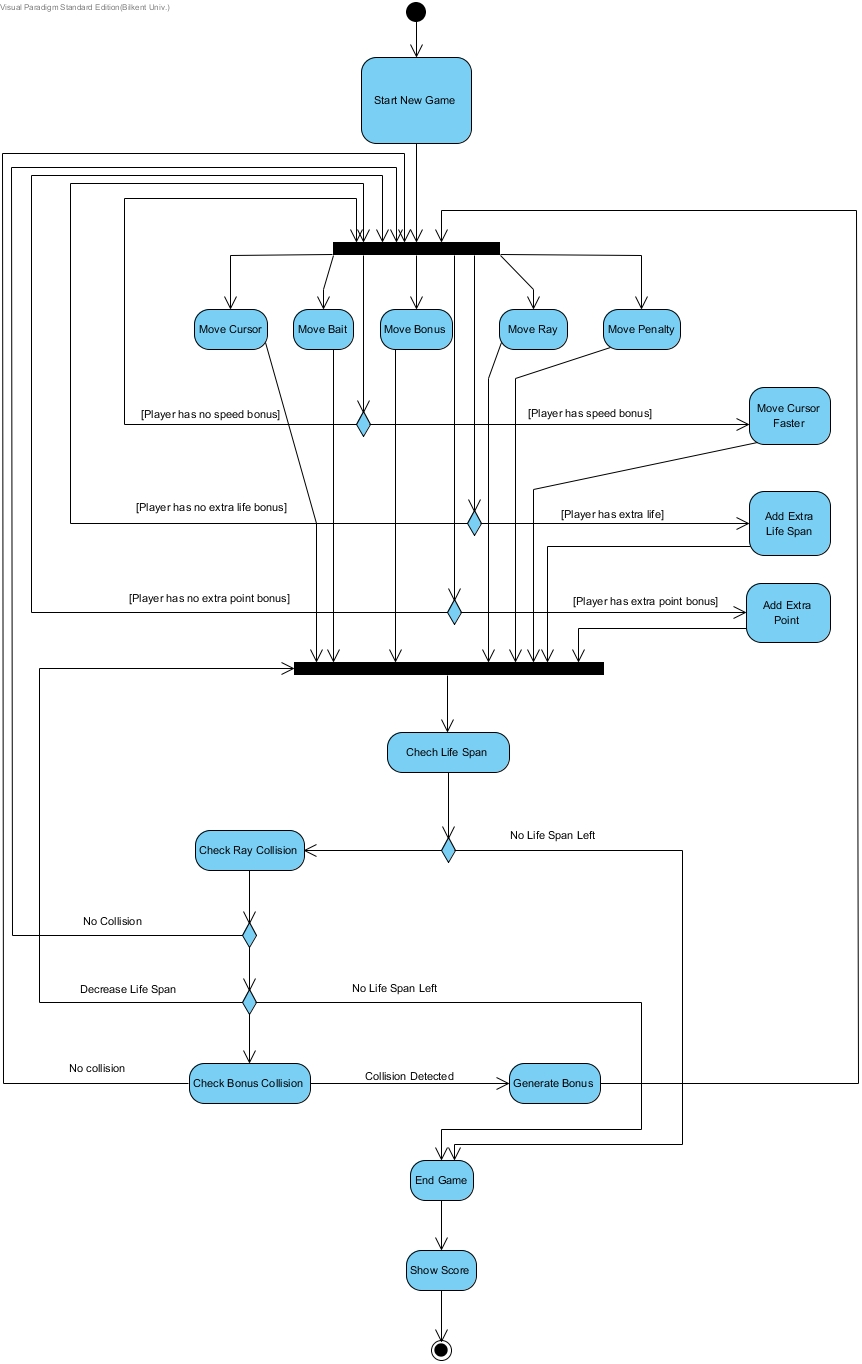


Figure 22: Activity Diagram of the game

When the user selects Play Game option from the Main Menu, the system initializes the game features, objects and crates the game scenario. When the preparation part is done the system waits for the user to interact with the mouse in order to start the cursor moving. When the user starts to move the cursor the system checks the game area to understand whether there are any baits or other objects falling from the top.

From this point, the system acts according to different scenarios until user’s lifespan is completed. When a collision is detected, the system decides next operation according to type of collision. For instance, if the cursor collides with bait, the player points increase and the game continues. However if the cursor collides with a bomb the player loses one life. Moreover if all of the lives of the player are lost the game stops and unfortunately the player loses the game. Then the system shows the score of player, finally.

If the detected collision is between the cursor and bonuses system applies bonus features to game status. In other cases system simultaneously updates game objects and plays collision sounds according to colliding game objects. Moreover, if player still has life and there is at least one life left, system maintains game loop and look for next collision. If the player still has life and he/she has collected at least 10,000 points, system provides access to the next level.

### 3.2.2 Sequence Diagrams

#### 3.2.2.1 Start Game

**Scenario:** Defne requests the System to start game by pressing the icon on desktop. After that System initializes a full screen window and opens the main menu page that has buttons for activities such as starting the game. Then the Defne clicks the start game button. Now the System creates the game engine to perform the show. After that game engine calls the file manager and it starts setting the game by getting the settings, getting images, getting scores and also the determining the level of the game. System then loads current scene by reading its object distribution from file manager, which just consists cursor, ray, player, bait and penalty objects (rock and bombs). Lastly, system enters the game loop in order to continuously update the game.

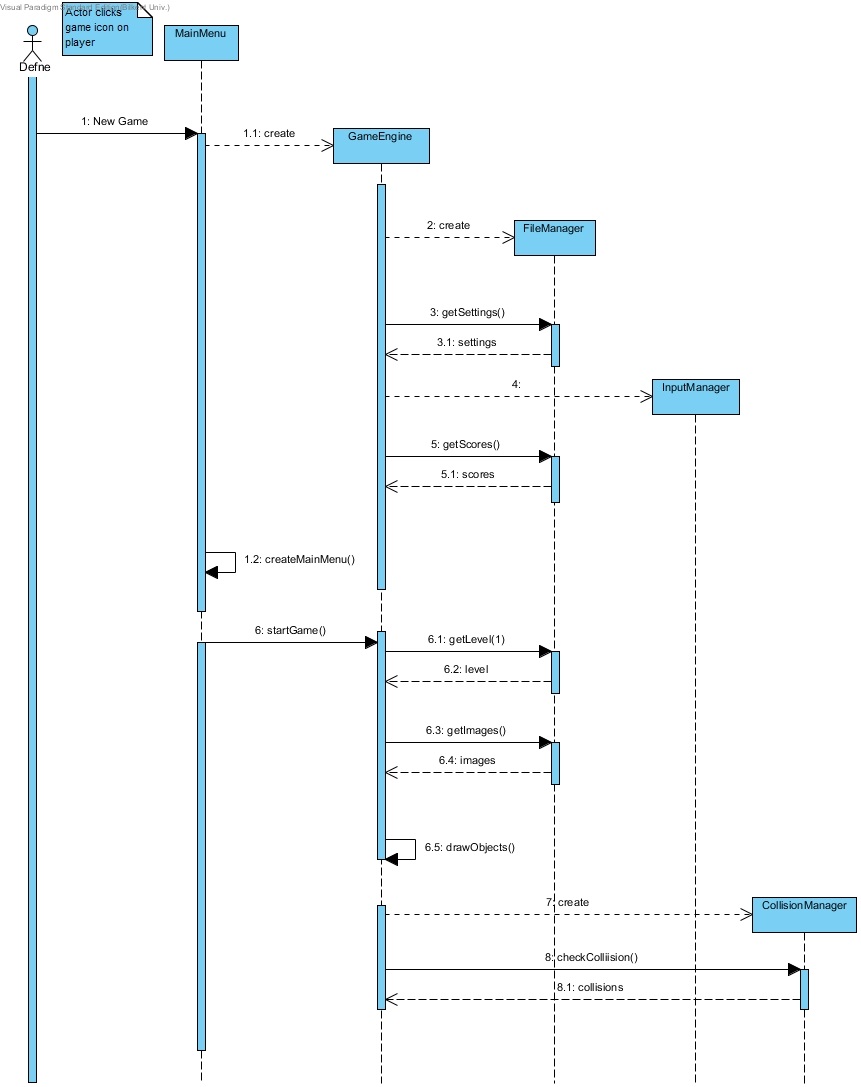
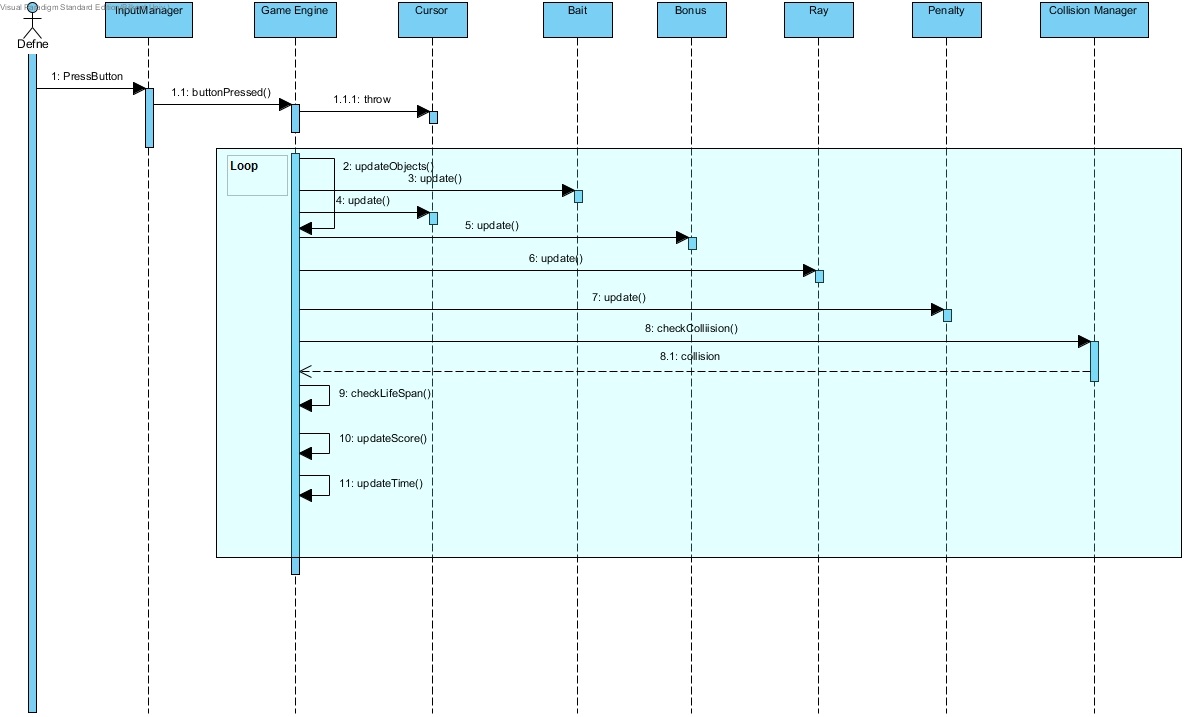


Figure 23: Sequence diagram which explains start game scenario

Here MainMenu is a boundary object by which user can perform some activities like starting game, viewing credits, changing settings. The locations of the game objects are stored in FileManager class, and it is responsible for the organization of the game screen.

#### 3.2.2.2 Play Game

Figure 24: Play game sequence diagram

**Scenario:** When Defne starts the game by selecting the play game choice the system initializes the game by instantiating game engine first. Then the GameEngine calls the FileManeger and they together set the settings for the game. When this preparation process is finished system waits Defne to interact with mouse to start the game. Firstly Defne starts the game by moving the cursor right and left to catch the baits. However, there could also be penalty objects falling from the above. If there is rocks or bombs that collide with the cursor, system checks the Defne’s number of lives to make a decision between two alternatives which are continuing game and ending game. If Defne has no life, game is over. If Defne has life, the GameEngine contacts with the CollisionManager to check whether there is a collision. Then if there is collision the GameEngine updates the life span of the player and it may decrease the time left for the Defne to finish the level. When a collision is detected, the system decides next operation according to type of collision. The time and life span damage that is caused by bombs are much more effective than that of rocks.

Moreover, if the cursor collides with bait the user gains points and that affects the possibility of him to pass the level. Other than the baits, the user can hit some bonuses which could produce alternative scenarios as having extra speed, gaining extra points or even increasing the life span. When any of these options happen GameEngine updates all of the objects and also contacts with the CollisionManager and produces the new settings according to the collision type that the CollisionManager decides.

#### 3.2.2.2 Change Settings

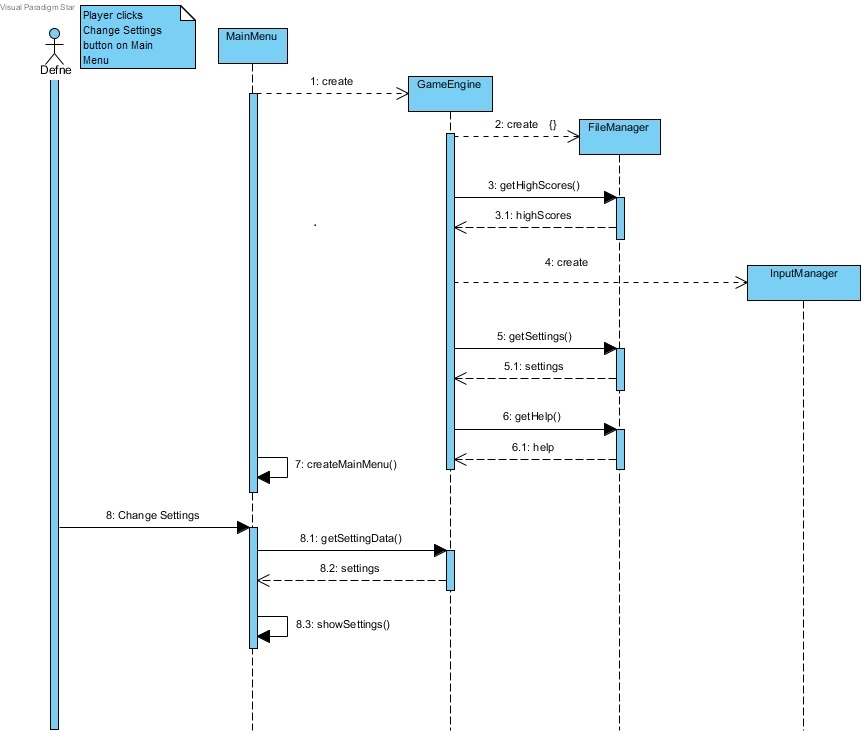


Figure 25: Change Settings Sequence Diagram

**Scenario:** Defne decided to change the game settings. She presses change settings button from the MainMenu object, the System displays settings, and the Defne changes the settings regarding his desires. After that, when Defne presses back button on the settings menu, the GameEngine updates the settings of the game to the new ones.

Here, Defne, as usual, contacts the System via the MainMenu object, and sends the command, change settings. Then, the MainMenu object contacts with the GameEngine object so that the command is done. After that, the GameEngine works with the FileManager simultaneously to create the changes in the settings as the Defne goes back to the playing the game. Moreover, in order to make some of the changes the GameEngine could also work with the InputManager. After this process is done the Main menu requests the GameEngine to return the settings and the GameEngine provides the settings that are changed. Finally the MainMenu shows the changed settings to the Player.

# 4. Design

## 4.1 Design Goals

**Non-functional Requirements**

**Adaptability:** Our program needs to work regardless of the operating system which the user is trying to launch Survivor. Thus, we have decided to implement our system using Java. Since Java® can work in all JRE installed platforms, the player would not have to worry about the adaptability of Survivor.

**Efficiency:** The game survivor will be having high efficiency and thus, low response time. In order to increase our system’s efficiency, we implement our source codes with low coupling and high coherence. In order not to disturb the player our game will run at least 40 fps.

Extendibility: In order for our game to be persistent through time, it is crucial to provide new features and components. In order to maintain the excitement of users our system will be extendable to have new bonuses and bait types.

**Usability:** Survivor will be a game which is user friendly and easy to understand. In order to achieve these qualities we design our interfaces such that the player would not need to have any prior knowledge of the gameplay. However, the game would have increasing difficulty in its levels.

**Modifiability:** To be able to handle any alterations of the system functionalities, we will implement our system with low coupling of subsystems. Thus, our system would be easy to modify.

**Tradeoffs**

**Efficiency – Reusability:**

Regarding the reusability, since we would not use our system to compose another system, the classes are designed specifically for the tasks of our game so the implementation would not be more complex than necessary.

**Functionality – Usability:**

To have more and different players interested, the interface of the game would be plain and not too complicated. Thus, the functionality will be basic. With regard to our main intentions of implementing Survivor, that is to entertain the player, we shall focus more on the usability of the system. Therefore, there will be plain menus and simple features in order to have the user enjoy the game.

## 4.2 Sub-System Decomposition

In this section, the system is divided into independent parts to clarify how it is organized. We choose three-tier architecture to design the system because it is the most suitable design pattern that fits our system structure. Diagram describing of our design is depicted below.

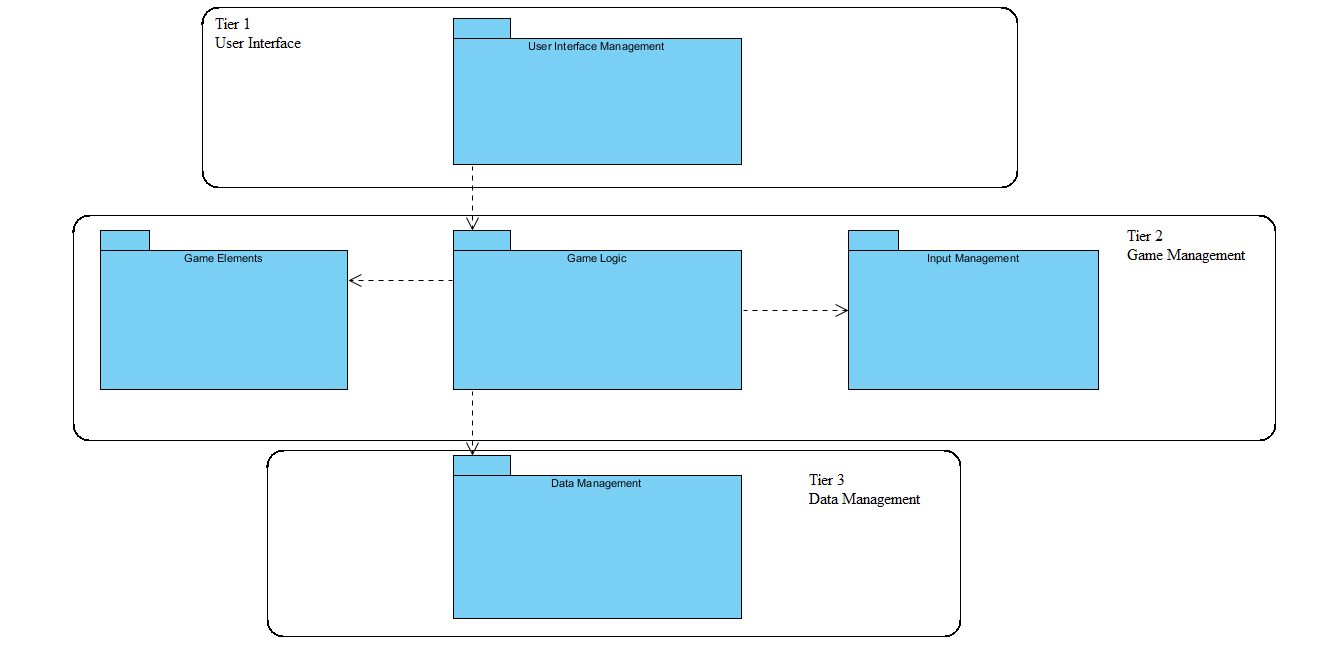


Figure 26: Basic Subsystem Decomposition

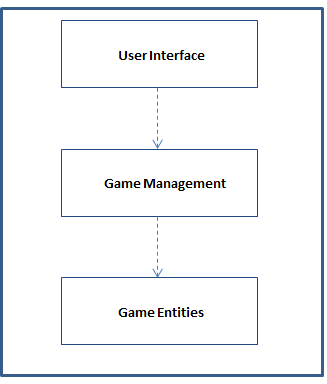


Figure 27: Layers of System

In this presentation layer, we have classes responsible for providing an interface between user and the system. The class “Main Menu”, which is in the “User Interface Management” package, is the point where interaction starts between user and program. It transmits user’s choices to “Game Logic” package. After this transmission, user’s choices will be evaluated by the “Game Logic”. If user chooses play game, “Game Engine” constructs the game layout with the help of the “Game Visualizer” class. In every stage of the game, “Game Logic” package will be in interaction with “Input Management” package to evaluate users’ inputs.

“Game Elements” package is responsible for providing the game layout. The information in this package will be updated according to results of the game objects’ interactions.

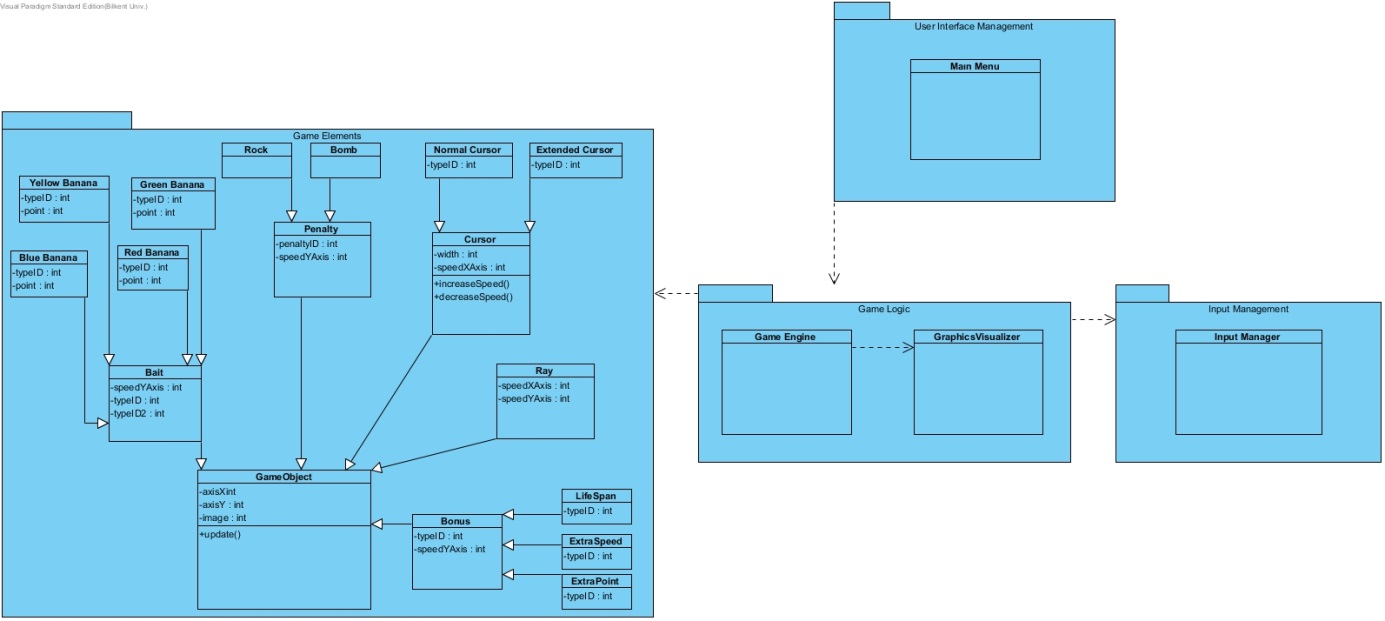


Figure 28: Layer 1 and Layer 2

“Data Management” package contains the “File Manager” class, which deals with file operations and data entities. “File Manager” will save and load the game data by interacting with the file system. The game objects and their attributes are located in the “Game Elements” package. This package stores all information about the game objects such as speed of objects, current positions and images.

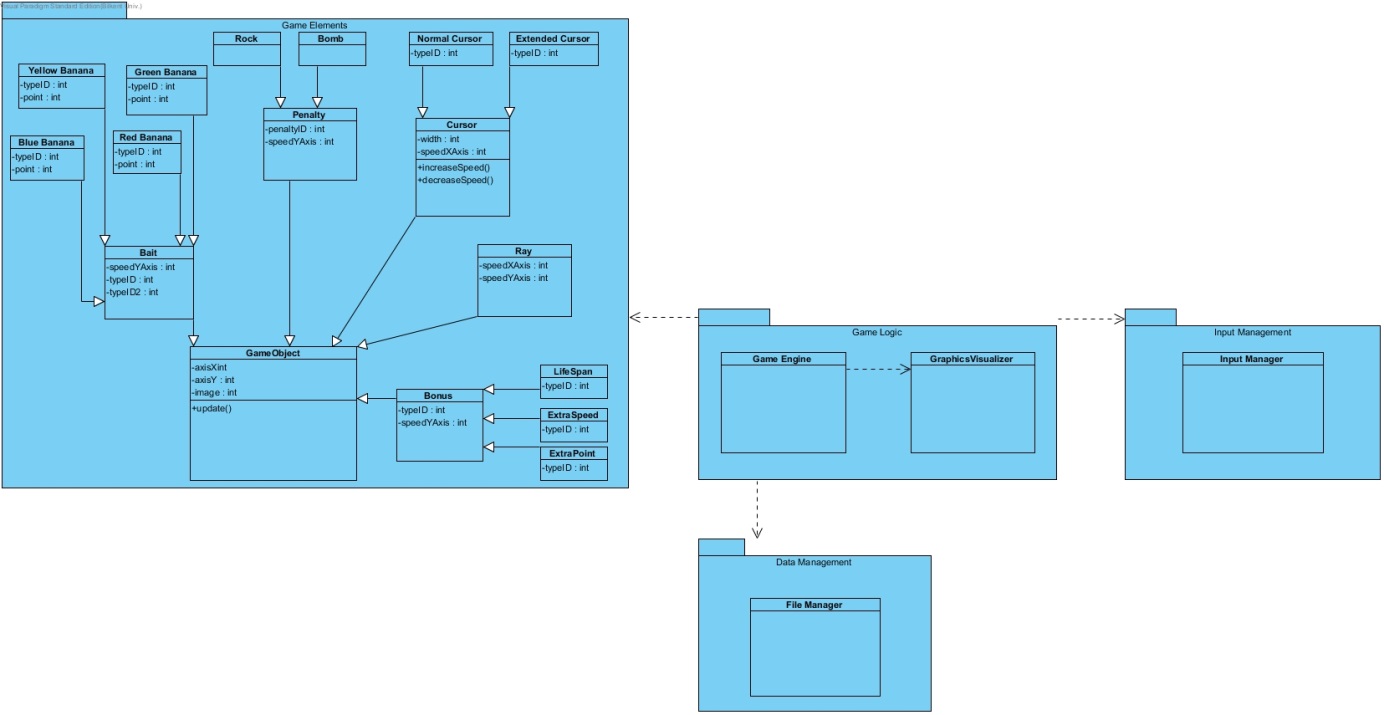


Figure 29: Layer 2 and Layer 3

User Interface Management Subsystem has 4 classes that are responsible for providing an interface for interaction between user and system. “Main Menu” class is the façade class of this subsystem. “Settings Panel”, “View Panel” and “Pause Menu” are used to display other submenus such as Settings, Help, Credits, High Scores and Pause Menu.

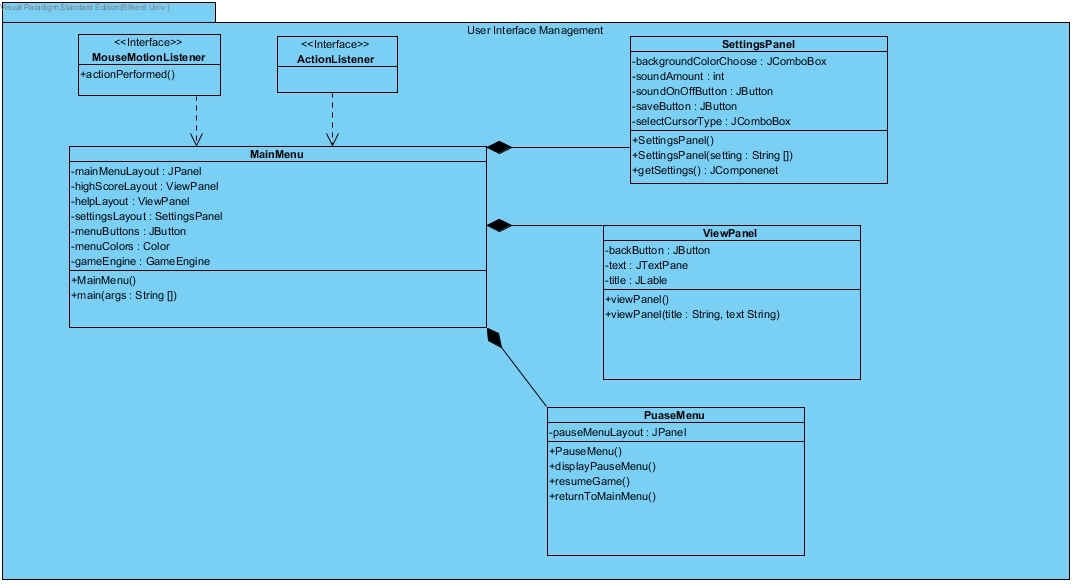


Figure 30: User Interface Management Subsystem

Game Logic package is responsible for game mechanics such as collision and display. Display management is handled by the “GraphicsVisualizer”, collision, detection and other game mechanics are handled by “Game Engine” class. Also, “SoundManager” class handles with the sound management of the game.

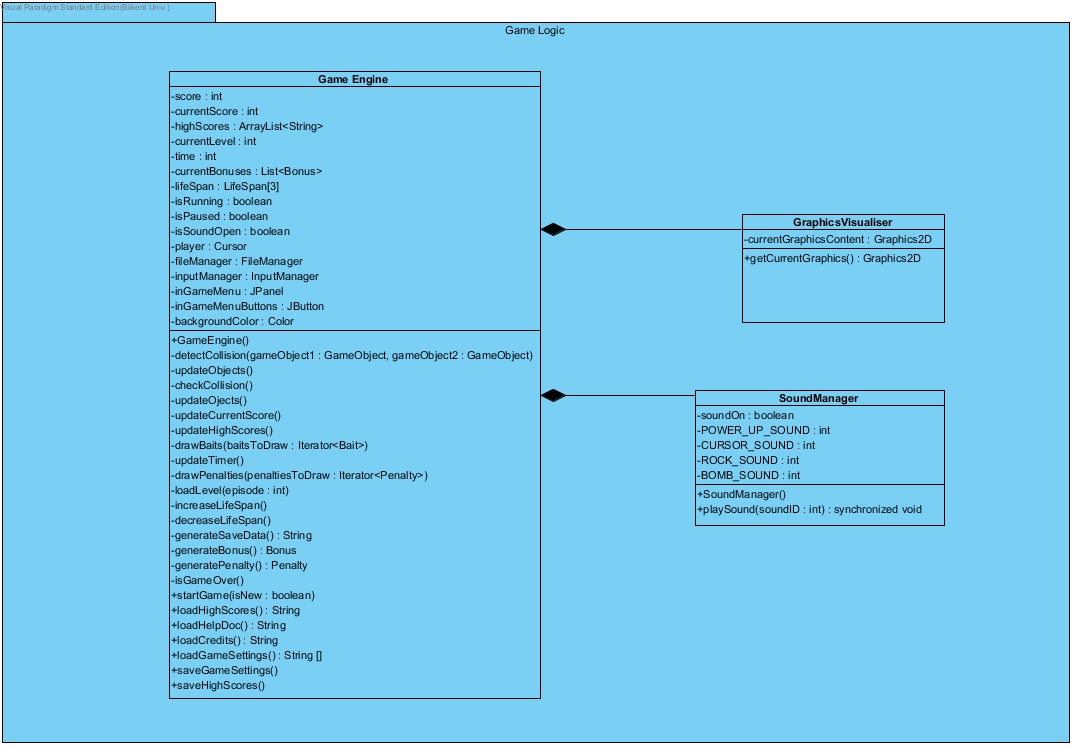


Figure 31: Game Logic Subsystem

“Input Management” package is responsible for the inputs required to control system. It detects user’s inputs and transmits these inputs to other subsystems to process.

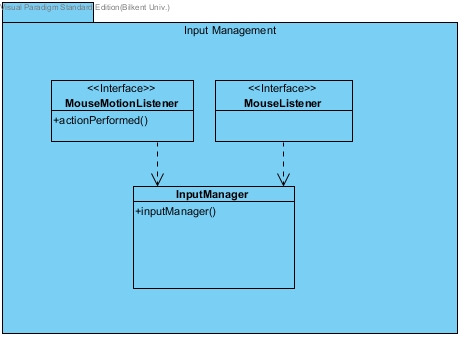


Figure 32: Input Management Subsystem

“Game Elements” subsystem provides objects to show on the screen. It has essential objects of game such as Cursor, Bait, Bomb, Rock and several Bonuses. Also in this package, there is Enumeration which is CursorState.

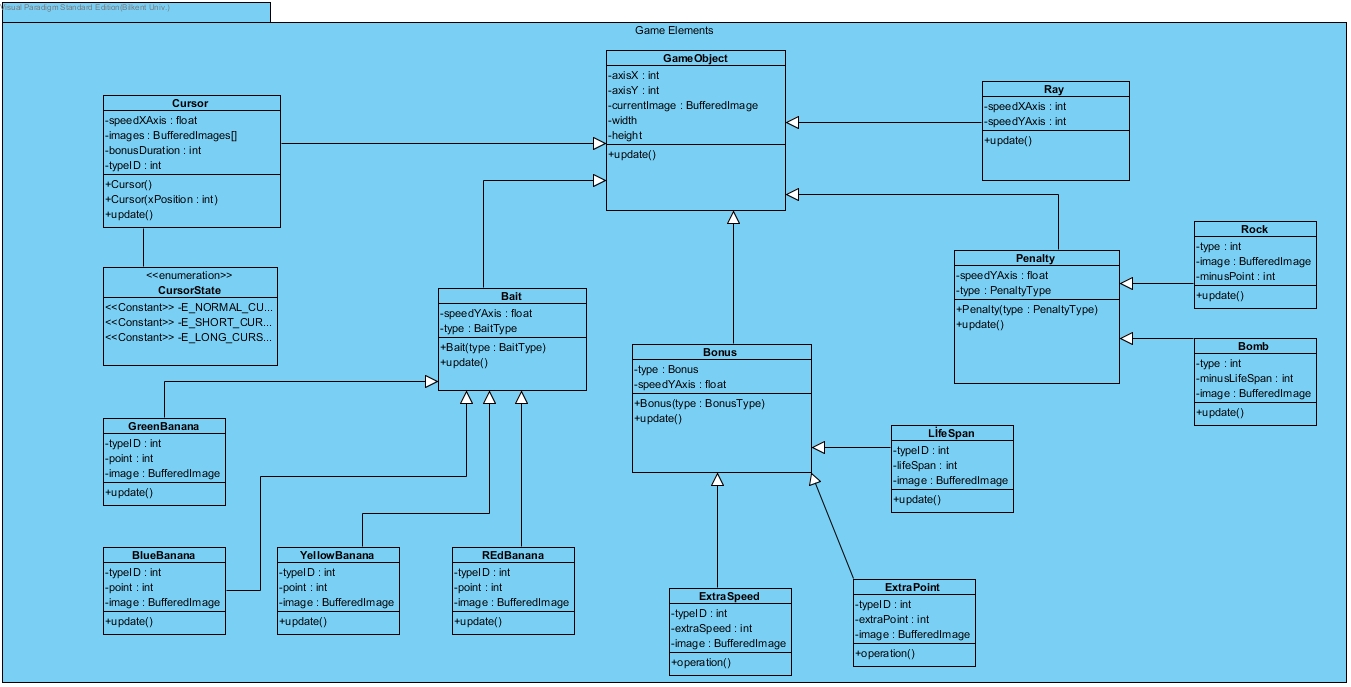


Figure 33: Game Elements Subsystem

“File Management” subsystem is used for saving and loading operations of the game. These operations are handled by the “FileManager” class of this subsystem.

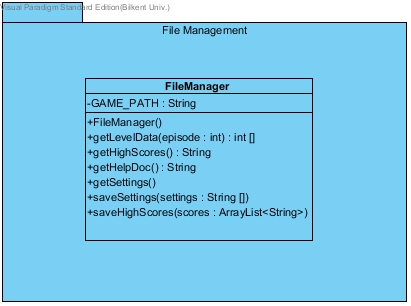


Figure 34: File Management Subsystem

## 4.3 Architectural Patterns

### 4.3.1 Layers

In our system decomposition we have decomposed our system into three main layers. These are View, Game Management and Data. These three layers are decomposed as hierarchical while grouping the subsystems that provide related services. The View layer is the highest hierarchy. It has User Interface Management subsystem and it is responsible for the interaction with the user. The following layer which is Game Management has Game Elements, Game Logic and InputManagement subsystems in it. It is responsible for bringing together and controlling the entity objects. Finally, our third layer is Data and it has Data Management subsystem in it. It is responsible for controlling and managing the data of our game such as scores and player lists. Our layer decomposition also proposes the closed architectural style, in which a layer can only access to the layer below it.

### 4.3.2 Model View Controller

In this architectural structure, main approach is to choose the three subsystems named Model, View and Controller. As we have divided the subsystems into three parts, we have isolated the domain knowledge from the user interface by adding a controller part between them.

In Survivor, we grouped our domain objects into the Data layer which constitutes the model of our system. Only the access of Manager Class is permitted by the system and these Manager classes are under the Game Management layer that constitutes the controller part. We grouped the classes which are responsible for providing the interaction between user and system into the View layer. It constitutes the View part since it communicates with the model part (Data) via controller part (Game Management). We have decided to use the MVC design since, by this architecture it is achieved that changes on the interfaces do not change the model of our system.

## 4.4 Hardware/Software Mapping

Our system requires Java Runtime Environment to be executed since we are using Java. As for the hardware requirements, since the player needs only mouse to control de game, considering mouse only will be enough. In terms of I/O requirements and graphical requirements, the game doesn’t require an advance computer, computers at today’s standards is enough to play the game. Moreover, if computer has GPU, that is a plus for the player while acceleration is available.

## 4.5 Addressing Key Concerns

### 4.5.1 Persistent Data Management

The data of our system shall be stored in the hard disk or memory that is provided by the user. Since the data should be accessible in real time there should not be any external databases. Thus our system stores the persistent data on the users memories as txt files. All of the images that our system will use would be stored unencrypted to encourage modifying them. For instance, bait images, bomb image and background images.

### 4.5.2 Access Control and Security

Due to the fact that Survivor does not require the players to be authenticated to play the game, the system would not have any kind of database to store the user credentials. Regarding the security matters our system will only give the data accesses to the GameEngine since it our main logic class.

Moreover, to control our system efficiently we made our crucial variables constant. Also, we have decentralized our systems game logic by delegating some simple tasks such as input management and image controlling of the game objects. By doing this we did not only reduce the complexity of the GameEngine class but also increase the robustness of the source code of Survivor.

### 4.5.3 Global Software Control

In the Global control flow the system waits for the user to interact with the game, regarding its Model View Controller architecture. This interaction can be done by simply moving the Mouse. Input Management Subsystem listens to the events of Mouse and make the GameEngine aware of the changes. After that GameEngine updates the view of the game with warning the other subsystems. Since we aim to have a better performance, we develop our system with smaller subsystems and divide the workload of the classes. This also helps us prevent the bottlenecks.

### 4.5.4 Boundary Conditions

The system would give error if its saved data or the file they are saved is corrupted. Moreover it will delete those data. The system would return to the maın menu screen if all the lives of the player are gone. The system would upgrade the high scores when the game is over. The system would update the high scores if the player has broken a record. When the game is over the system will show the high scores after updating them regarding the players score. Then when the user decides the game will return to the main menu. For this version of Survivor there are finite number of levels so if the player completes all of the levels, again high scores will be updated and the game will return to the main menu.

# 5. Object Design

## 5.1 Pattern applications

**a) Adapter Pattern**

 The adapter pattern is a [design pattern](https://en.wikipedia.org/wiki/Software_design_pattern) that allows the [interface](https://en.wikipedia.org/wiki/Interface_(computer_science)) of an existing [class](https://en.wikipedia.org/wiki/Class_(computer_science)) to be used from another interface. It is often used to make existing classes work with others without modifying their [source code](https://en.wikipedia.org/wiki/Source_code). Thus, we can use this pattern to make our source code reusable.

In this pattern, an adapter helps the software engineer use two incompatible interfaces to work together. The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

Since Java does not allow us to make multiple inheritances, we can use the Object Adapter pattern. That is, we will have a middle interface that would use inheritance to be qualified of the traits of our actual İnterface classes (the middle interface will extend the real interface class) and it would use delegation to have access to the system class’ (an engine class) attributtes and data.

**b) Façade Pattern**

The  façade pattern is a  [pattern](https://en.wikipedia.org/wiki/Software_design_pattern) that uses an object that provides a simplified interface to a larger body of code. A facade can make a class package or class pile easier to use, understand and test. Moreover, since the facade has convenient methods for common tasks, make the library more readable, reduce [dependencies](https://en.wikipedia.org/wiki/Coupling_(computer_programming)), thus allowing more flexibility in developing the system.

Since Facade design pattern is often used when a system is very complex or difficult to understand and that we cannot provide the source code to the user, we could use the benefits of Facade design. This pattern hides the complexities and provides a simpler interface to the user. Thus, we thought that we could make a more simple interface that covers the basic options of the system and shows only them to the player. Moreover the system will act more complex according the choice of options that the user sellects.

**c) Composite Pattern**

In [software engineering](https://en.wikipedia.org/wiki/Software_engineering), the composite pattern is that does partitioning. The composite pattern means that a group of objects is to be treated in the same way as a single instance of an object. The intent of this pattern is to compose objects into hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

In our system, we have several types of bonuses and penalty objects. Therefore, we could use Composite Pattern to group them as one type and then specialize them with their own attributes. Since composite should be used when the user ignore the difference between compositions of objects and individual objects this pattern could be applied to our system. Composite would be a good choice since it is less complex in this situation to treat primitives and composites as homogeneous.

## 5.2 Class Interfaces

**User Interface Management Interface**

User interface management subsystem provides graphical components of our system. This subsystem also manages transitions between panels.

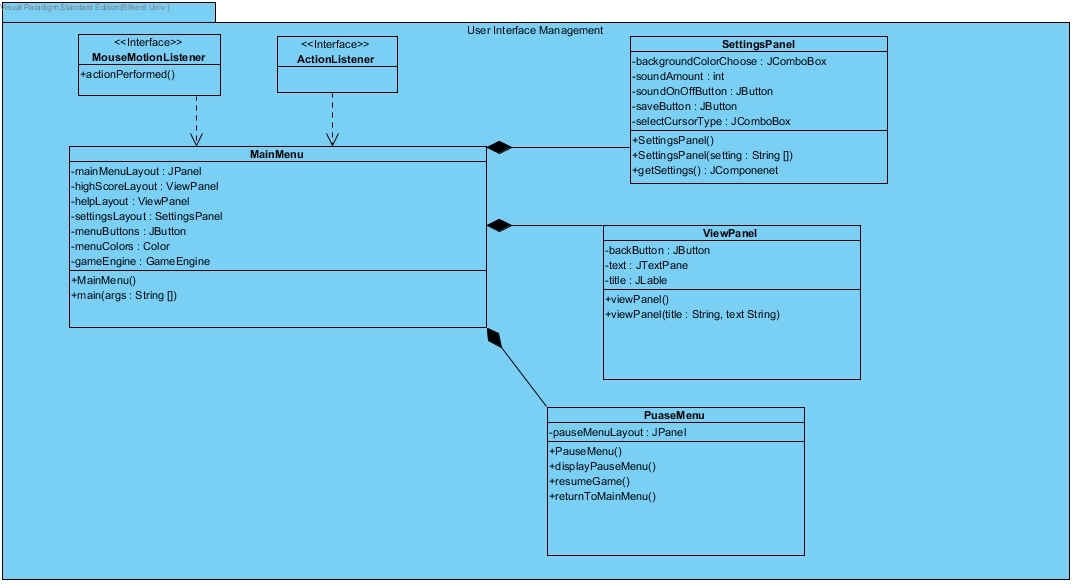


Figure 35: User Interface Management Subsystem

**Main Menu Class**

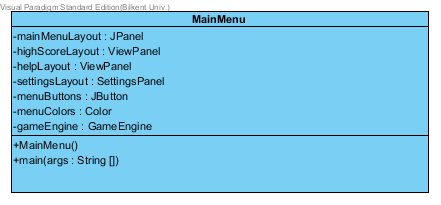
****

Figure 36: Main Menu Class

**Attributes:**

**private JPanel mainMenuLayout:** mainMenuLayout used in graphical user interface to show Main Menu of the game on screen.

**private ViewPanel highScoreLayout**: highScoreLayout used in graphical user interface to show High Scores of the game on screen. This layout is constructed by using ViewPanel class with its components like labels, text pane and buttons etc.…

**private ViewPanel helpLayout**: helpLayout used in graphical user interface to show Help Menu on screen. This layout is constructed by using ViewPanel class with its components like labels, text pane and buttons etc.…

**private SettingsPanel settingsLayout**: settingsLayout used in graphical user interface to show Settings Menu on screen. This layout is construcred by using SettingsPanel class and its components like buttons, label, combo boxes etc….

**private JButton menuButtons**: menuButtons used in graphical user interface to show Menu Buttons on screen. User can manage interface by using this buttons.

**private GameEngine gameEngine**: GameEngine provides reference to game logic subsystem.

**Constructors:**

**public MainMenu**: MainMenu initializes mainMenuLayout, highScoreLayout, helpLayout, settingsLayout, menuButtons and gameEngine.

**Methods:**

**public static void main(args String[]):**This method is main method of the game which initializes all components when program runs.

**Settings Panel**

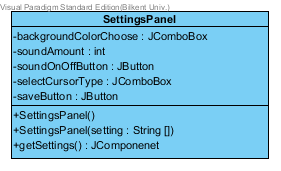
****

Figure 37: Settings Panel Class

**Attributes:**

**private JComboBox backgroundColor**: backgroundColor used in Settings Panel to provide different background colors for users.

**private int soundAmount**: soundAmount used in Settings Panel to modulate sound in the game.

**private JButton soundOnOffButton**: soundOnOffButton used in Settings Panel to indicate whether sound is on or off.

**private JComboBox selectCursorType**: selectCursorType used in Settings Panel to provide different cursor types for users. Cursor type can be arranged by using this property.

**private JButton saveButton**: saveButton used in Settings Panel to confirm changes.

**Construnctors:**

**public SettingsPanel():** It initializes backgroundColor combo box, soundAmount, soundOnOffButton button, selectCursorType combo box and saveButton button.

**Methods:**

**public SettingsPanel(String settings[]):** returns a current settings of game.

**public JComponent getSettings():** returns a previous settings of game.

**View Panel**

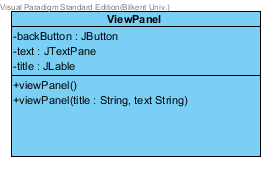
****

Figure 38: ViewPanel Class

**Attributes:**

**private JButton backButton**: This button used in View Panel. User can turn back previous page by using this button.

**private JTextPane text**: text used in View Panel to provide area to print strings such as help document or credits in it.

**private JLable title**: title is used in View Panel to print title of panel.

**Constructors:**

**public viewPanel(String title, String text):** It takes title and text as parameter to construct panel.

**Pause Menu Class**

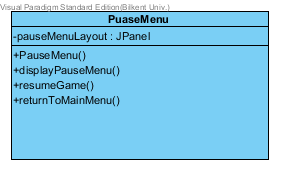
****

Figure 39: Pause Menu Class

**Attributes:**

**private JPanel pauseMenuLayout**: pauseMenuLayout used in graphical user interface to show Pause Menu on screen.

**Constructor:**

**public PauseMenu():** It initialize instances of Pause Menu object.

**Methods:**

**public void diplayPauseMenu**(): This method adds pauseMenu panel on frame.

**public void resumeGame**(): This method invokes game logic subsystem to resume game.

**public void returnToMainMenu**(): This method end current game and add MainMenu panel on frame.

**Game Logic Subsystem Interface**

In this subsystem our controller objects are grouped to manage game logic. We have 3 components which are GameEngine, GraphicsVisualizer and SoundManager classes in this subsystem. SoundManager handles with the user input and GameEngine and SoundManager classes handle logic of the game.

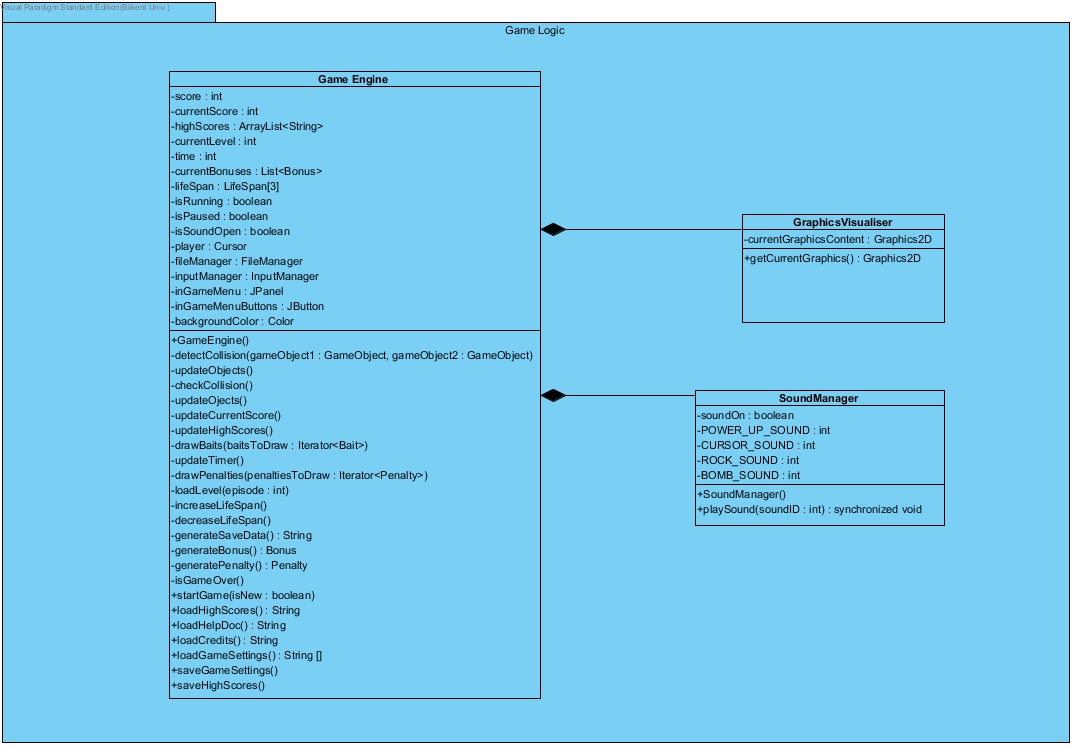


Figure 40: Game Logic Subsystem

**Game Engine Class**

****

Figure 41: Game Engine Class

**Attributes:**

**private int currentScore**: This value keeps the player’s score. When player catches bait or penalty, this value updates immediately.

**private ArrayList<String> highScores**: This array list keeps information about players’ who has highest 10 score with their scores.

**private int currentLevel**: This value keeps current level information about the game.

**private int time**: This value keeps time information about the game.

**private List<Bonus> currentBonuses**: This list keeps bonus information about the game, since there can be different kinds of bonuses in our system, this value is needed to create different types of bonuses such as extra life span, extra point etc.

**private LifeSpan[3] lifeSpan**: This value keeps how many life span is user has. This value equal 3 initially. When user catches extra life span bonus or hits bomb penalty, this value updates +1 or -1 respectively.

**private boolean isRunning**: This flag is true if game is running. According to value of this flag, game engine create new object such as bonuses or penalties.

**private boolean isPaused**: This flag is true if game is paused.

**private boolean isSoundOpen**: This flag is true if game sound is open.

**private Cursor player**: This object references Cursor class to play game.

**private FileManager fileManager**: This object references File Manager class to get saved info from files such as settings or high scores.

**private InputManager inputManager**: This object references Input Manager class to play game by using mouse.

**private JPanel inGameMenu**: This panel creates game menu panel.

**priate JButton inGameMenuButton**: This panel creates game menu buttons.

**private Color backgroundColor**: This value determines background color of the game.

**Constructors:**

**public GameEngine**(): initializes GameEngine object with default attribute values.

**Methods:**

**public void detectCollision(GameObject gameObject1, GameObject gameObject2):** determines whether there is a collision between cursor and other objects(baits, penalties) or not.

**public void updateObjects**(): updates every object in the game.

**public void checkCollision**(): checks collision between game objects.

**public int updateCurrentScore**(): updates current score in the game according to catched objects.

**public ArrayList<String> updateHighScores**(): If player gets higher score than tenth highest score, this class updates high scores list and return new list.

**public void drawBaits(Iterator<Bait> baitsToDraw):** draw baits with given attribute to play game.

**public int updateTimer**(): updates time of game every second.

**public void drawPenalties(Iterator<Penalty> penaltiesToDraw**): draw penalties with given attribute to play game.

**public void loadLevel(int episode)**: loads back level when game start again.

**public int increaseLifeSpan**(): If user catches extra life span bonus, this class updates number of life spans of user by 1.

**public int decreaseLifeSpan**(): If user catches bomb penalty, this class decreases number of life spans of user by 1.

**public String generateSaveData**(): This method generates data to save such as highest scores or current settings of the game.

**public Bonus generateBonus**(): This method generates bonuses by determining the type of them.

**public Penalty generatePenalty**(): This method generates penalties by determining the type of them.

**public boolean isGameOver**(): This method return true if user has zero life span. This means game is over.

**public void startNewGame(boolean isNew**): This method create new game when user clicks new game button.

**public String loadHighScores**(): When user clicks high scores button, this method reads high scores from file and prints them on screen.

**public String loadHelpDoc**(): When user clicks view help button, this method reads help document and prints information on screen.

**public String loadCredits**(): When user clicks credits button, this method shows our names on screen.

**public String[] loadGameSettings**(): This method load game settings by reading them file and starts game with these settings.

**public void saveGameSettings**(): If user changes current settings of game, this method saves them to file to use again.

**public void saveHighScores**(): This method saves high scores information to file.

**Graphics Visualizer Class**

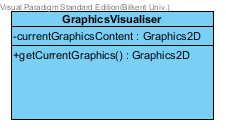
****

Figure 42: Graphics Visualizer Class

**Attributes:**

**private Graphics2D currentGraphicsContent**: creates currents graphics of the game.

**Methods:**

**public Graphics2D getCurrentGraphics**(): visualizes graphics of the game.

**Sound Manager Class**

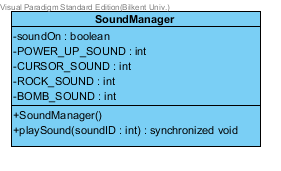
****

Figure 43: Sound Manager Class

**Attributes:**

**private boolean soundOn**: this attribute is a flag to determine whether the sound is enabled or disabled in game.

Constant static variables in this class are designed to map the proper sound samples as id’s.

**Constructor:**

**public SoundManager**(): initializes the SoundManager. Sound of game is initially off.

**Methods:**

**public synchronized void playSound(int soundID):** this method is invoked GameEngine class when needed. It plays sound sample according to given value.

**Game Elements Subsystem Interface**

Game elements subsystem holds domain specific objects of the game. In this subsystem, we have 16 classes. Our game objects inherit from an abstract base class called GameObject. GameObject class performs proper actions on game entities such as creating and modifying them. In Figure 10 game elements subsystem is visualized. In this section, each class of this system will be explained.

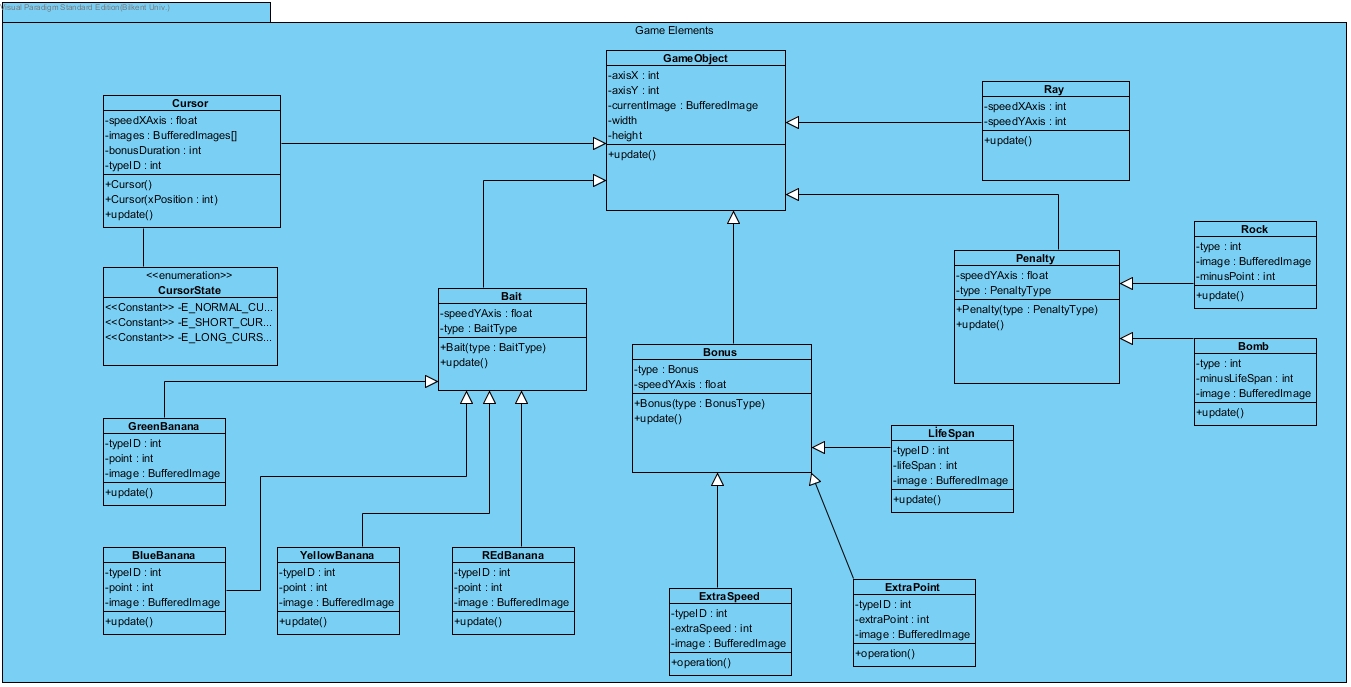
****

Figure 44: Game Elements Subsystem

**Game Object Class**

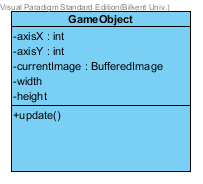
****

Figure 45: Game Object Class

**Attributes:**

**private int axisX**: X axis of position of object.

**private int axisY**: Y axis of position of object.

**private BufferedImage currentImage**: currentImage is an image of object which will be updated accourding to other attributes of the class.

**private int width**: Width of the object.

**private int height**: Height of the object.

**Methods:**

**public void update**(): updates the axisX and axisY attributes of the game object.

**Penalty Class**

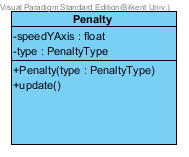
****

Figure 47: Penalty Class

**Attributes:**

**private float speedYAxis**: determines the velocity of fall of penalty objects.

**private PenaltyType** type: provides information about type of penalty object.

**Constructor:**

**public Penalty(int type):** creates penalty object with given attributes.

**Methods:**

**public update**(): updates Y axis of object.

**Rock Class**

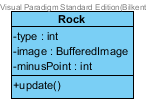


Figure 46: Rock Class

**Attributes:**

**private int type**: determines type of object.

**private BufferedImage** image: determines image of the object.

**private int minusPoint**: determines how many point is user going to lose if collision detects between rock object and cursor.

**Methods:**

**public void update**(): update Y axis of object.

**Bomb Class**

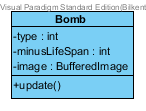
****

Figure 48: Bomb Class

**Attributes:**

**private int type**: determines type of object.

**private int minusLifeSpan**: determines how many life span is user going to lose if collision detects between bomb object and cursor.

**private BufferedImage** image: determines image of the object.

**Methods:**

**public void update**(): update Y axis of object.

**Ray Class**

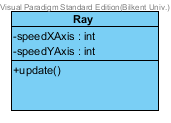
****

Figure 49: Ray Class

**Attributes:**

**private int speedXAxis**: determines X direction velocity of ray.

**private int speedYAxis**: determines Y direction velocity of ray.

**Methods:**

**public void update**(): update Y axis of object.

**Bonus Class**

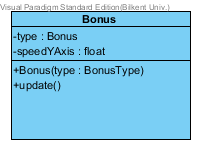
****

Figure 50: Bonus Class

**Attributes:**

**private int type**: determines type of bonus object.

**private float speedYAxis**: determines speed of object in Y direction.

**Constructor:**

**public Bonus(BonusType type):** determine type of bonus object and constructs bonus object according to given attributes.

**Methods:**

**public void update**(): update Y axis of object.

**Life Span Class**

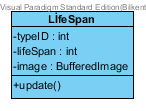
****

Figure 51: Life Span Class

**Attributes:**

**private int typeID**: determine type of bonus object.

**private int lifeSpan**: determine how many life span is user going to win when user catches this bonus object.

**private BufferedImage** image: determines image of the object.

**Mehtods:**

**public void update**(): update Y axis of object.

**Extra Point Class**

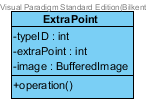
****

Figure 52: Extra Point Class

**Attributes:**

**private int typeID**: determines type of bonus object.

**private int extraPoint**: determines how many points is user going to win when user catches this bonus object.

**private BufferedImage** image: determines image of the object.

**Methods:**

**public void Update**(): update Y axis of object.

**Extra Speed Class**

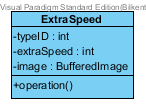
****

Figure 53: Extra Speed Class

**Attributes:**

**private int typeID**: determines type of bonus object.

**private int extraSpeed**: determines how much speed is cursor gain when user catches this bonus object.

**private BufferedImage** image: determines image of the object.

**Methods:**

**public void Update**(): update Y axis of object.

**Bait Class**

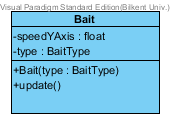
****

Figure 54: Bait Class

**Attributes:**

**private float speedYAxis**: determines speed of bait object in Y direction.

**private int type**: determines type of bait.

**Constructor:**

**public Bait(int type):** determine type of bait and constructs bait according to determined type.

**Methods:**

**public void update**(): update Y axis of object.

**Red Banana Class**

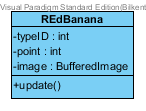
****

Figure 55: Red Banana Class

**Attributes:**

**private int typeID**: determine type of banana object.

**private int point**: determine how many point is user going to win when user catches banana.

**private BufferedImage** image: determine image of object.

**Methods:**

**public void update**(): update Y axis of object.

**Yellow Banana Class**

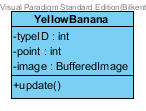
****

Figure 56: Yellow Banana Class

**Attributes:**

**private int typeID**: determine type of banana object.

**private int point**: determine how many point is user going to win when user catches banana.

**private BufferedImage** **image**: determine image of object.

**Methods:**

**public void update():** update Y axis of object.

**Blue Banana Class**

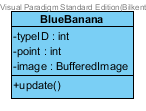
****

Figure 57: Blue Banana Class

**Attributes:**

**private int typeID**: determine type of banana object.

**private int point**: determine how many point is user going to win when user catches banana.

**private BufferedImage** image: determine image of object.

**Methods:**

**public void update**(): update Y axis of object.

**Green Banana Class**

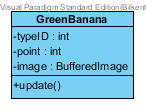
****

Figure 58: Green Banana Class

**Attributes:**

**private int typeID**: determine type of banana object.

**private int point**: determine how many point is user going to win when user catches banana.

**private BufferedImage** image: determine image of object.

**Methods:**

**public void update**(): update Y axis of object.

**Cursor Class**

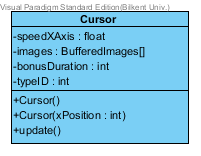
****

Figure 59: Cursor Class

**Attributes:**

**private float speedXAxis**: determines speed of cursor object in X axis.

**private BufferedImage image**: determines image of cursor object.

**private int bonusDuration**: determines duration of cursor bonus or cursor penalty.

**private int typeID**: determines type of cursor. There are different types of cursors in the game. When user catches long cursor bonus, width of cursor will increase. If user catches short cursor penalty, width of cursor will decrease.

**Constructor:**

**public Cursor**(): construct cursor.

**Methods:**

**public Cursor(int xPosition):** construct cursor with given attribute.

**public void update():** update X axis of object.

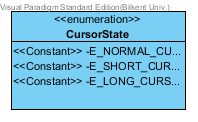
****

Figure 60: Type of cursor object.

**Input Manager Subsystem Interface**

This subsystem is designed to detect user’s actions which are performed by mouse. In this context, this subsystem is implemented proper interfaces of java.

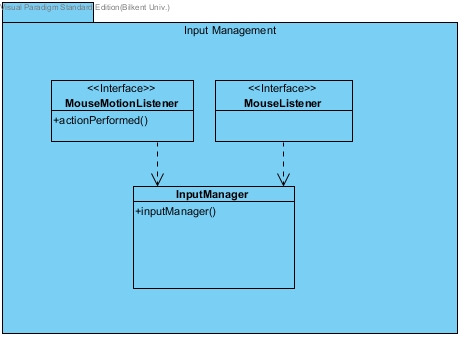
****

Figure 61: Input Manager Subsystem

**Input Manager Class**

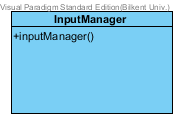
****

Figure 62: Input Manager Class

**Constructor:**

**public inputManager**(): constructs input manager to detect user actions.

**File Management Subsystem Interface**

File Management Subsystem is used in all saving and loading operations of the game by interacting with the file system. These operations are handled by the "FileManager" class of this subsystem.

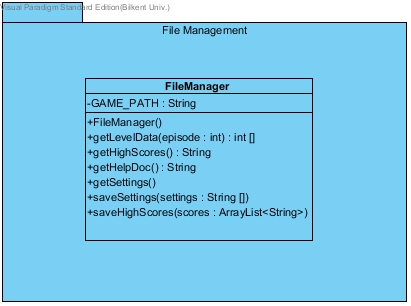


Figure 63: File Management Subsystem

**File Manager Class**

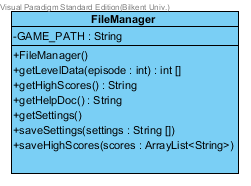


Figure 64: File Manager Class

**Attributes:**

**private String GAME\_PATH**: determines the game path to save data.

**Constructor:**

**public FileManager**(): constructs file manager.

**Methods:**

**public int[] getLevelData(int episode**): This method is used in new game operation by getting corresponding episode information.

**public String getHighScores():** sends high score information such as user name and point of user to game engine.

**public String getHelpDoc():** sends help document information to game engine to displayed on screen.

**public String[] getSettings**(): sends previous settings information to game engine. If user doesn’t change game settings, game will be played by using previous settings.

**public void saveSettings(String[] settings):** saves new settings of the game into the file.

**public saveHighScores(ArrayList<String> scores):** saves high scores of the game into the file.

## 5.3 Specifying Contracts

For the benefit of both our system and the user, the system class methods shall warn the user about its acceptable and unacceptable input values or types, and their meanings and also return values or types, and their meanings. Survivor shall have error and exception conditions that can occur, and shall enlighten the user about their meanings.

Since we use Model View Controller attributes, all class relationships are between user interface classes and engine classes (GameEngine). An interface class is obliged to make calls to its corresponding engine class features where the resulting state of the engine class is not violated by the interface call. Subsequently, the engine classes are obliged to provide a return states and data that does not violate the state requirements of the interfaces, whilst the game is played. For instance, GameEngine class data buffer may require that data is present in the buffer when a delete feature is called. Subsequently, the GameEngine guarantees to the interface that when a delete feature finishes its work, the data item will, indeed, be deleted from the buffer. Java also requires this attribute by its Garbage Collector.

As for the class invariants it shall be said that the state of the class (any class that is used) will be maintained at the end of each feature execution.

Regarding the correctness of the Survivor, if the class invariant and preconditions are true before an engine class is called by and interface class, then the invariant and the post-condition will be true after the service. Moreover, when making calls to an engine class, the system should not violate the engine class' preconditions.

# 6. Conclusions and Lessons Learned

In conclusion, in this report, we aimed to create an action game called Survivor. Our report has parts such as, requirements specification, System Model, Subsystem Decomposition, Detailed Object Design and we also include a part that we clarify our contracts.

In requirements specification, we examined mostly all of the possible requirements which any player could have performed. Moreover, we have indicated our functional and nonfunctional requirements regarding these performances.

In system model, our report consists of the following sections:

1. Use case model

2. Dynamic models

3. Class model

4. User interface

Firstly, we have decided our use case scenarios. Then, in the second part, System Design, we have made our sequence diagrams and activity diagrams. In our sequence diagrams we tried to demonstrate the possible actions that the player could take. Activity diagram indicates mainly our game play. It represents the actions of our game components, cursor, baits, bonuses and penalty objects. Moreover, our class diagram represents our implementation.

Moreover, ın the Subsystem Decomposition part, the detailed class diagram is depicted in order to provide a better understanding about the interactions and basic fundamentals of our software. We have divided our class diagram into subsystems or in other words packages. With the help of this class diagram, the detailed subsystem descriptions were made easier to comprehend.

In the Detailed Object Design part we have commented and clarified the attributes of our classes and interfaces that we have decomposed in the Subsystem Decomposition part. Furthermore, we have showed our classes in Class Diagrams and as particular and ındividual class diagrams.

In addition we have included a contracts part to clarify our contracts of Survivor. In brief, programming by contract prescribes that software designers should define [formal](https://en.wikipedia.org/wiki/Formal_methods), precise and verifiable interface specifications for [software components](https://en.wikipedia.org/wiki/Component-based_software_engineering#Software_component), which extend the ordinary definition of [abstract data types](https://en.wikipedia.org/wiki/Abstract_data_type) with [preconditions](https://en.wikipedia.org/wiki/Precondition), [post-conditions](https://en.wikipedia.org/wiki/Postcondition) and [invariants](https://en.wikipedia.org/wiki/Invariant_(computer_science)). These specifications are referred to as "contracts”. Moreover, The Programming by contract approach assumes all interface components that invoke an operation on a server component will meet the preconditions specified as required for that operation.

Furthermore, this project has been a successful and meaningful assignment for us, on our path to be Computer Engineers. We have seen that in order to make a time wised efficient and user friendly software, we have to make preparatory work and preliminary examinations of the design and clients needs. In addition we have seen that if we use design patterns it is easier for us to change our software attributes without changing all of our source code. Last but not least, since we have used the Model View Controller attributes, we have learned that it is best for software engineers since it also allows us to make changes in the view (GUI) of the system without changing the algorithms.

To sum up, Survivor is a game that we have decided to develop from the idea of DXBall. However, the game’s visuals and features will be unique to its attributes.