CS 319 - Object-Oriented Software Engineering

Project Final Report

Feeding Bobby

Group 1-K:

Arda Kıray

Gülce Karaçal

Volkan Sevinç

Okan Şen

Table of Contents

Table of Figures

1. Introduction

Feeding Bobby is a progressive single-player game for computers in which the user will start off as a little fish. Throughout the span of the game, the main goal of the player will be to get bigger by eating smaller fish, to survive by avoiding from bigger fish and to defeat the final boss while facing some mid game bosses.

Feeding Bobby will provide an environment in which the user will encounter some power ups such as freeze time, speed boost, vitality and hostile fishing rods which will immediately kill the character. In addition to these, we will allow the player to alter its physical appearance by choosing from various skin options.

The game progresses by finishing levels on the world map. As the game progresses, it gets harder as well by sending more threats in the player's way. The player will build up scores by evolving and eating. The final score of each chapter will be converted and added to the user's in-game currency that can be used to buy various skin options.

Feeding Bobby is a local game on a Windows desktop PC. The player won't be interacting with the game other than controlling the mouse. The character will pursue its path according to the way the player uses the mouse.

2. Current System

Not available.

3. Proposed System

3.1 Overview

The game consists of many levels spread out on a world map. Each of these maps will have different tasks to succeed but basically the main task is not to be eaten

while eating other fishes. With each fish eaten our score will increase and on specific score limits player's fish, Bobby, will get bigger in size.

By getting bigger, the character will evolve into bigger species. In each level, there will be two types of fish except the player whose sizes are respectively smaller and bigger than Bobby. As Bobby grows bigger by eating smaller fish, its size will increase enough to eat the bigger fish and then the player will be faced with even bigger fish. The growth rate of Bobby will depend on the type and the number of fish eaten.

Each level will have different tasks to accomplish in order to finish them. Some will want the user to reach a score and some will want the user to eat a specific fish. These tasks will get harder to accomplish with the increasing progression of the player.

In each level, the user will start by eating smaller fish and build up points to reach the next level of growth which is evolving in the game. For example, when the player eats enough fish and reaches 500 points Bobby will evolve and grow in size. This will give an extra life to the player as well.

There is a combo system which will let the player grow faster with each successive eaten fish by multiplying the points taken but the combo will be over once the player takes damage. The game also encourages to keep the combo on and evolve as many times as possible by giving points for evolving too and this evolving score will be multiplied higher by the combo number. Additionally, there will be a frenzy bar which will fill as the player eats fish. It will only increase by eating fish and will only be decreased by enabling the frenzy mode, taking damage will not affect this bar. Frenzy mode will activate once its bar is full and it'll decrease till it is empty. During the frenzy mode Bobby can eat any fish whether it is bigger, smaller or special fish that would normally kill Bobby. Anything eaten will give points.

Bobby will die if he is bit by any bigger fish in any level's beginning until he evolves by eating enough fish. Then Bobby will be able to be bit twice, first one decreasing his size to the former size and second one killing him as if in the beginning of the game. To make things easier for the user after being bit, Bobby will be

invulnerable for a few seconds to get back to a safe spot on the screen and continue eating smaller fish.

After when a level is finished, whether by accomplishing the tasks or by being eaten, the final score will be added to the high scores table if it is high enough. There will be power ups, power downs, random events and special fishes in the levels.

3.1.1 Power-ups

Figure 1: Speed Boost

- **Speed Boost:** This power up will slow all the fish on the screen and any fish that will come on the screen except Bobby, for a short duration.

Figure 2: Freeze Time

- **Freeze Time:** This power up will freeze everything on the screen except Bobby.

-

- Figure 3: Vitality/Health Boost

- **Vitality/health Boost:** This will grow Booby once in size and let him evolve.

- Figure 4: Kill Bigger Fish

- **Kill Bigger Fish:** As the name suggests this power up will diminish all the bigger fish on the screen immediately.

- Figure 5: Little Fish Swarm

- **Little Fish Swarm:** This power up will spawn and send a little fish swarm on the screen, possibly helping Bobby to have a boost on score and combo.

3.1.2 Power Downs

Figure 6: Slow

. **Slow:** This will make all the other fish on the screen and any other fish coming in screen faster except Bobby.

Figure 7: Stun

- **Stun:** This will stun Bobby for a short duration making him vulnerable for any incoming fish or random events.

- Figure 8: Health Decrease

- **Health Decrease:** This will immediately degrow Bobby and decrease his size once (de-evolve). This will not be available during first state of evolving of Bobby.

- Figure 9: Poison

- **Poison:** This will make Bobby sick in the stomach and he will not be able to eat any other fish for a short duration.

3.1.3 Insta-Killers

- **Fishing Rod:** A fishing rod will come down from the top of the screen on a random side. This rod will stay there for a short amount of time and will pull back. If Bobby is close enough to be pulled back too, he will die instantly.
- **Sea Mines:** These mines will be deeper in the ocean so in the first levels they will not be very effective. If Bobby hits one of these mines, he will instantly die.

3.1.4 Random Events

- **Sea Currents:** In some levels sea currents will be available, randomly happening, it will push Bobby in its direction. Swimming against this current will be slower while going with it will be a lot faster than normal speed.
- **Geysers:** Hot geysers will randomly spawn at the bottom of the screen and will channel for a few seconds, exploding a hot stream of water vertically in its wake. If Bobby gets hit by this stream of hot water he dies.
- **Tornado:** With a low chance, a tornado will spawn and the game mode will change. The tornado will stay locked on the left side of the screen and the screen will start moving to the right in a sidescrolling manner. Bobby will have to evade all the incoming obstacles and threats while staying away from the tornado. If Bobby gets caught by the tornado he will die.

3.1.5 Special Fishes

A) Friendly

- **Astronaut Fish:** Eating this will grow Bobby once in size by evolving him. Astronaut fish has an oxygen tank strapped on its back and Bobby loves oxygen!
- **Enraged Fish:** This fish will increase Bobby's rage and add boost to the frenzy bar.

B) Hostile

- **Pufferfish:** Eating this fish will cause Bobby's death. The pufferfish will explode in Bobby's stomach after a few seconds.
- **Drunk Fish:** Eating this fish will convert all the controls.
- **Jelly Fish:** Staying around these jelly fish will electrocute Bobby causing him to degrow if electrocuted for a few seconds.

3.2 Functional Requirements

* The user will be able to access the help menu that consists of information about how to play the game and descriptions of special power-ups.
* The player will be able to have some power-ups according to the type and the number of fish eaten.
* The player should be able to pause the game whenever he/she wants.
* The system should pause the game and should stop all movements when the pause button is pressed.
* The system saves the game after the player completes each level.
* The player should be able to load the game whenever he/she wants.
* The system should be able to show credits.
* The user can toggle music on and off.

* The player should be able to enter his/her name at the beginning of the game.
* The user should be allowed to play single player.
* The game should be controlled by the movements of mouse.

3.3 Non-Functional Requirements

* Control system of the game should be easy to understand and should feel natural.
* Concepts of the game should be easy to understand and react to.
* Game should not contain any game-breaking or progress preventing bugs.
* Load times should be minimal.
* Visuals such as menus should be easy to understand.
* Controls should contain the minimum delay possible.
* Concept art such as models and backgrounds should not harm or tire the player.

3.4 Constraints

* The game will be implemented in Java.
* The game will run on Windows desktop PC.
* Adobe Photoshop will be used for various designs such as fish and backgrounds.
* FL Studio will be used for custom music and sound effects.
* Player must have pre-installed Java 8 or higher versions in his computer

3.5 System Model

3.5.1 Use Case Scenarios

3.5.1.1 Start a New Game

Use Case Name: Start a New Game

Primary Actor: Player

Entry Condition: Player clicks “Start a New Game” button from Main Menu

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

Event Flow:

- A tutorial page displays onto the screen which contains all necessary information to prepare user for his first experience

- The main map displays on to the screen which contains a journey that consists of several chapters. Initially only the first chapter of the journey is available for the user and sequentially each next chapter will be opened after completing the previous one.

- Player selects the first chapter and game starts

- Player aims to eat smaller fishes to get bigger and complete the growth amount of the first chapter while simultaneously trying to survive.

- System keeps the growth amount of the user’s character on the screen as “Growth bar”

- System keeps the score of the user

- System automatically saves the current progress after each chapter

Pre-condition: For the first use, game settings are set as default. If user changes any of game settings, adjusted settings will be used throughout the journey until the user makes another change.

Post-condition: If the score of the user is eligible to record on high-score table, high- score table will be updated by the system.

Success Scenario Event Flow:

1. Game is started by the system
2. Player starts a new game
3. A tutorial screen displays
4. Player selects the first chapter of the game
5. Game starts and player plays until according growth amount is satisfied
6. Player finishes the current chapter and the map of journey displays

Player repeats finishing consecutive chapters until the final chapter is complete or player dies and return to the journey of map.

7. System records players gathered points to his in game currency and high-score table if the resulting score is eligible for high score table

Alternative Flows:

Player will face with various power-ups and threats during the game and encountering one of them can change the flow of events as:

1. Player collects a special power-up

2. System does the necessary changes for applying the power-up

3. The power-up vanishes due to time out or death of the character

3.5.1.2 Pause Game

Use Case Name: Pause Game

Primary Actor: Player

Entry Condition: Player selects the pause icon from the game screen

Exit Condition: Player re-selects the pause icon from the game screen

Event Flow:

- Player pushes the pause icon from the game screen

- System stops the game

- Player pushes the pause icon from the game screen

- System unfreezes the game

Pre-condition: User must be on the game screen

Post-condition: -

Success Scenario Event Flow:

1. While in game player selects the stop game icon
2. System freezes the game until player selects to unfreeze the game by using the same icon

3.5.1.3 Load Game

Use Case Name: Load Game

Primary Actor: Player

Entry Condition: Player selects the “Load Game” button from the Main Menu

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

Event Flow:

- Player selects the “Load Game” option from the Main Menu

- A list of auto-saved progresses displays on the screen

- Player chooses the desired one

- System loads the chosen saved progress’ game screen on the display

Pre-condition: There must be at least one saved game for loading that progress

Post-condition: The achieved progress will be over-written on to the same save file after loading game

Unsuccessful Scenario Event Flow:

1. Player selects the “Load Game” option from the Main Menu

2. A list of auto-saved progresses displays on the screen

3. Player chooses one to load

4. Due to corruption of saved data, game cannot be loaded and related error message displays on the screen

3.5.1.4 Change Settings

Use Case Name: Change Settings

Primary Actor: Player

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

Exit Condition: Player selects the “Return to Main Menu” button from Change Settings display

Event Flow:

- Player selects the “Change Settings” button from Main Menu

- A list of game settings displays on the screen

- Player adjusts desired settings from the list by filling the according boxes

- Player selects the “Return to Main Menu” button from Game Settings screen

- System automatically saves the preferred settings before returning to the Main Menu

Pre-condition: For the first use, game settings are set as default. If user changes any of game settings, adjusted settings will be used throughout the journey until the user makes another change.

Post-condition: Adjusted game settings are updated.

Success Scenario Event Flow:

1. Player selects the “Change Settings” options from Main Menu

2. A list of game settings displays on the screen

3. Player fills or unfills the desired option’s check box

4. Player clicks the “Return to Main Menu” button

5. System automatically saves the changed settings

6. Main Menu screen displays

3.5.1.5 View Help

Use Case Name: Help

Primary Actor: Player

Entry Condition: Player clicks the question mark symbol at the bottom right corner of the Main Menu

Exit Condition: Player selects the “Return to Main Menu” button from the help screen

Event Flow:

- A tutorial screen consists of all necessary instructions to play the game displays

Pre-condition: -

Post-condition: -

Success Scenario Event Flow:

1. Player clicks the question mark symbol at the bottom-right corner of the Main Menu

2. A tutorial screen consists of all necessary instructions to be able to play the game displays

3. Player clicks the “Return to Main Menu” button

4. System automatically saves the changed settings permanently until the events 1-4 occurs again

3.5.1.6 View High Scores

Use Case Name: High Scores

Primary Actor: Player

Entry Condition: Player selects the “High Score” button from Main Menu

Exit Condition: Player selects the “Return to Main Menu” option from high scores table page

Event Flow:

- A list of high scores table corresponds to each saved players displays on the screen

Pre-condition: -

Post-condition: -

3.5.1.7 Exit Game

Use Case Name: Exit Game

Primary Actor: Player

Entry Condition: Player selects the “Exit Game” button from Main Menu

Exit Condition: -

Event Flow:

- System terminates itself

3.5.2 Use Case Model

Below is our Use Case Diagram and Use Cases as described in *Section 3.5: Use Case Scenarios.* Our primary intention was to keep the diagram as comprehensible as possible for the client.

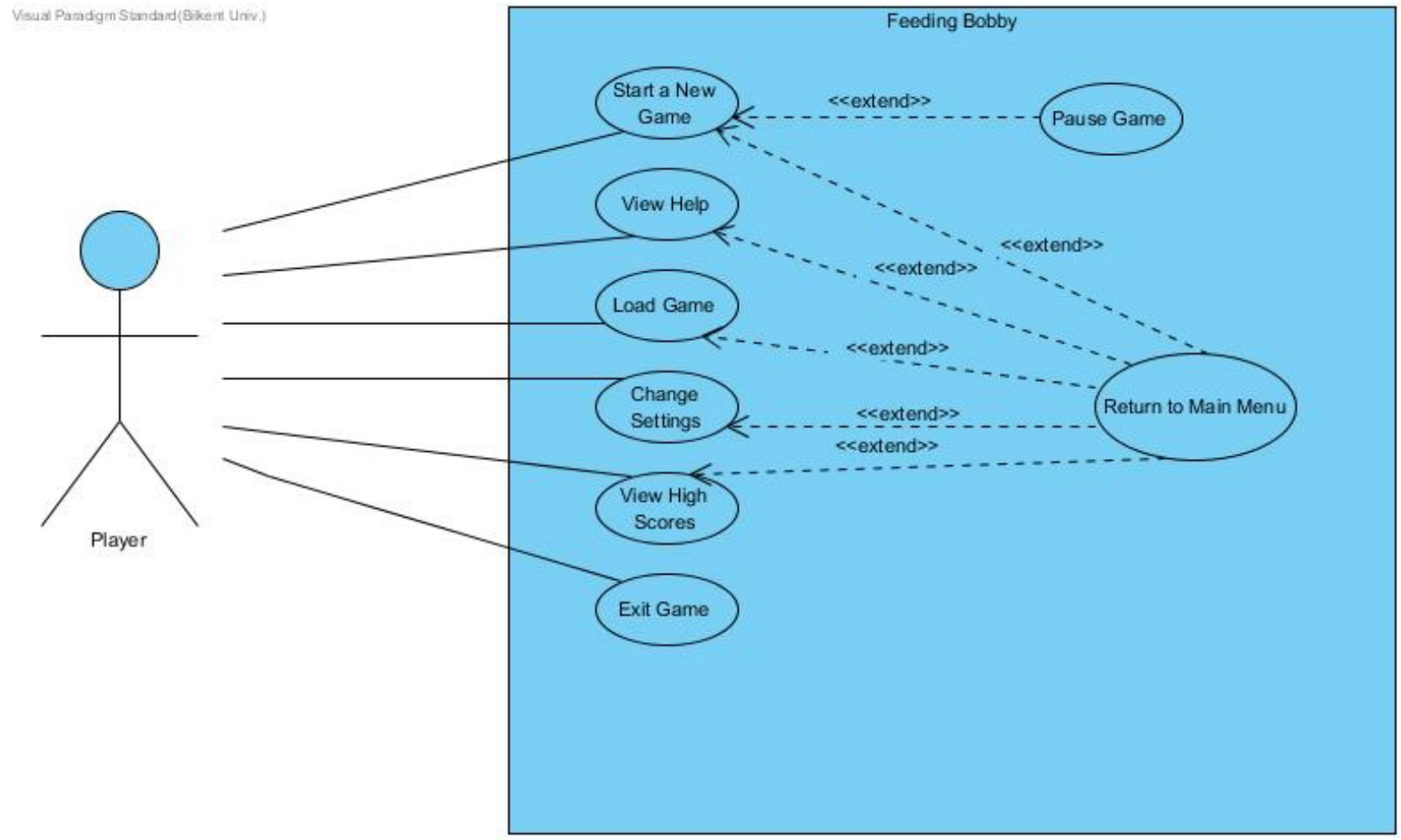


Figure 10: Use Case Model

3.5.3 Object Model

3.5.3.1 Class Diagrams

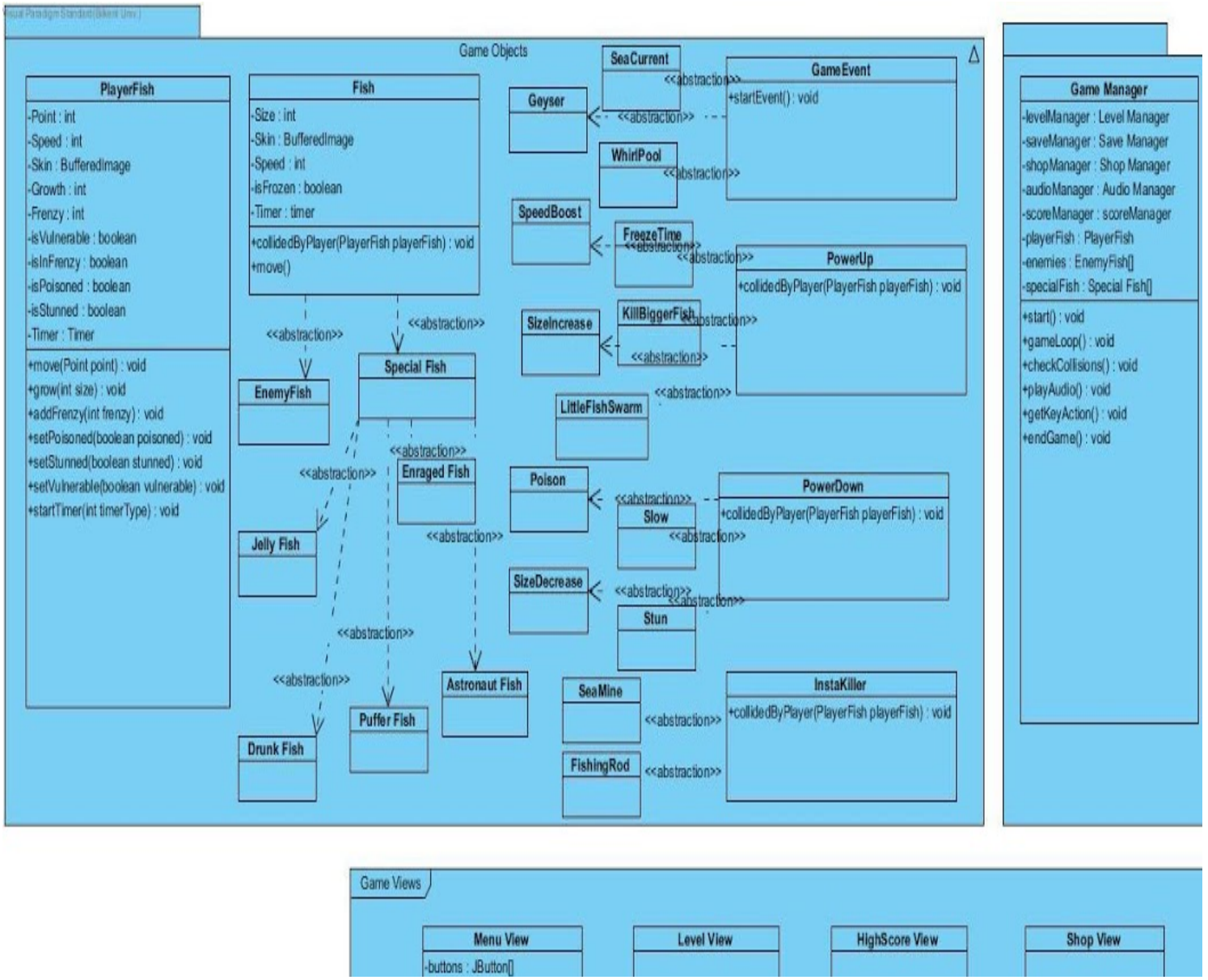


Figure 11: Class Diagram (Game Objects)

The above class diagram contains Game Objects and shows their relation. We have a Fish class which includes fish types that we have in our game such as Enemy Fish and Special Fish. PlayerFish class is the one containing attributes and operations of the user's fish. Moreover, we have PowerUp class including SizeIncrease and SpeedBoost etc. Similarly, we implement decreasing in power in PowerDown class which includes Slow and SizeDecrease etc.

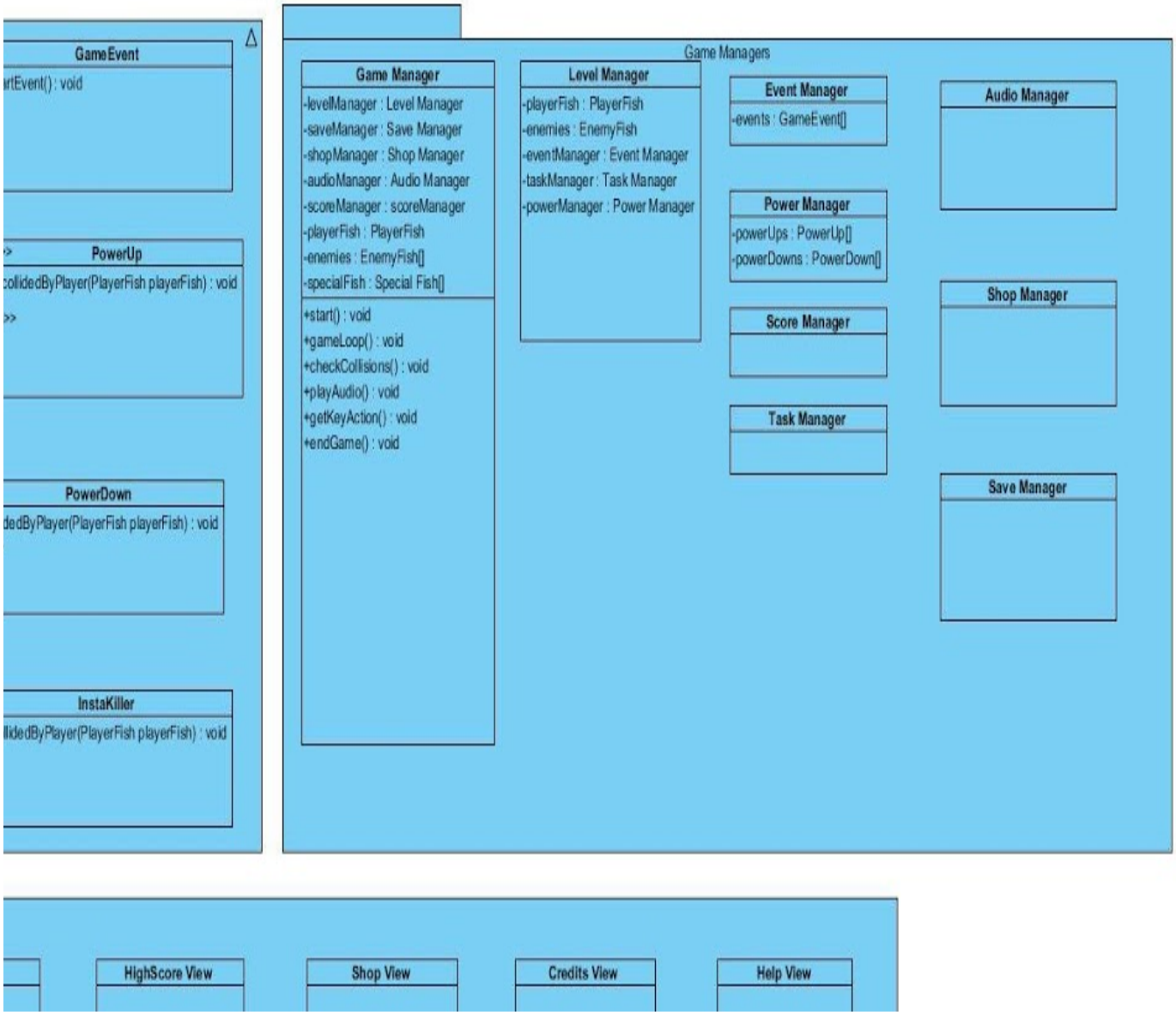


Figure 12: Class Diagram (Game Managers)

The above class diagram contains our Game Managers classes. We have decided to implement these classes separately because it is the easier to divide parts that implement crucial operations. GameManager class is the one where the main management of the game is handled. It manages the lifecycle of the game through working with other manager classes. FileManager class handles the connection

between the files associated with the game. Furthermore, we have other classes which manages the features of the game such as PowerManager, SaveManager etc.

3.5.4 Dynamic Models

3.5.4.1 Sequence Diagrams

3.5.4.1.1 Start Game

Considering the flow of events, the game is quite straightforward. In the main menu, we will provide five options to user, either to choose help button and go to the help screen that explains the game in basic steps, or to select shop to buy the item that is affordable such as a skin, or to choose change settings button, or to select load, or to click start button to start a new game. When the user goes to the help activity, there is no other option but to get back to the main menu. This is the same with other options except start and load game. This is the reason why we decided to create only crucial scenarios' sequence diagrams to avoid using unnecessary diagrams.

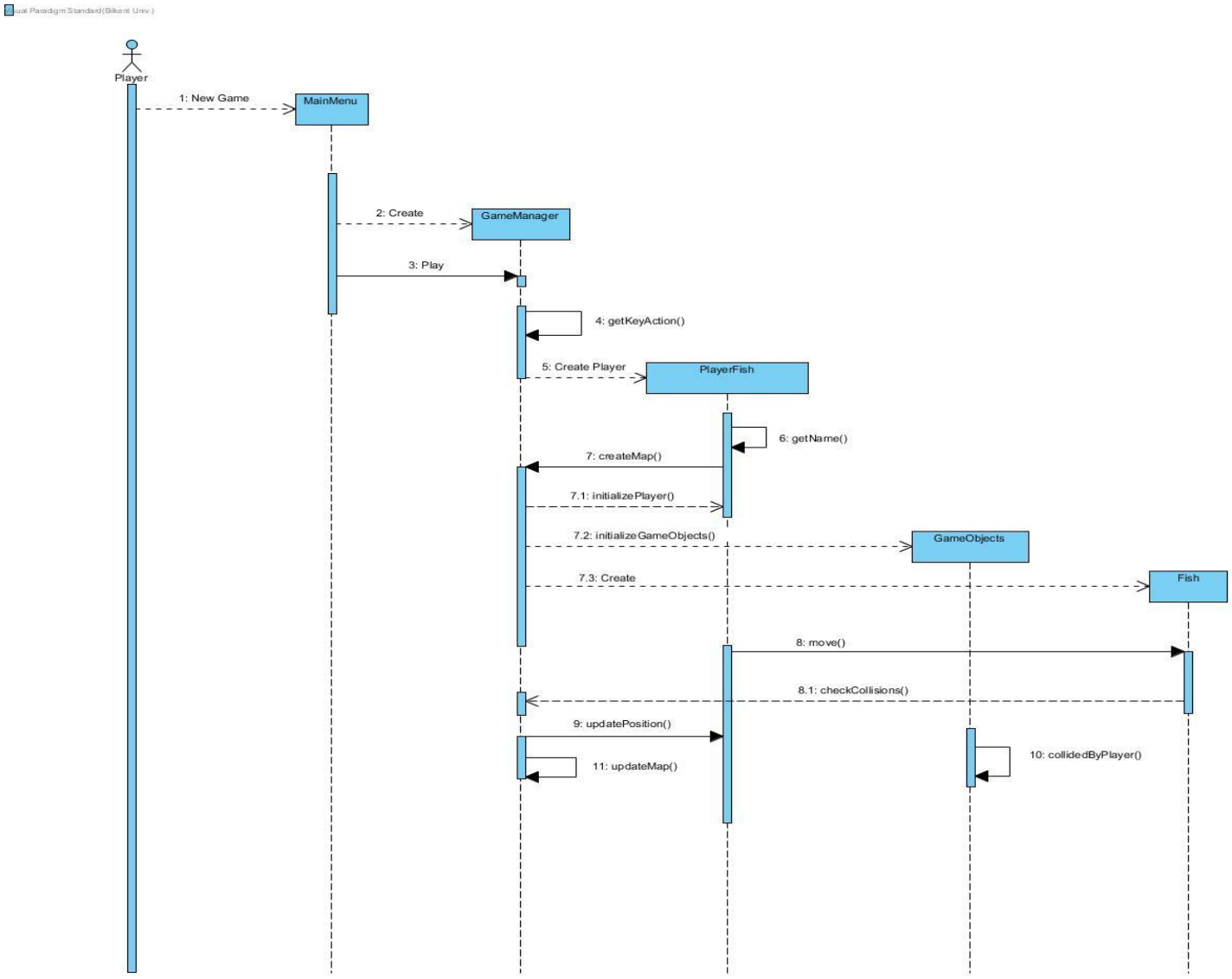


Figure 13: Start Game Sequence Diagram

Considering the above diagram, it can be said that the sequence begins when the user clicks on the game icon on the desktop. First of all, the main menu is reached and then the Game Manager class is created which is the one forming the Player Fish. And then, the user is required to enter a name. After entering the name, the map is created with all initial details including the start position of the player. Moreover, the position of the user is updated during the game.

After these steps, the fish is created and it returns the result of checking collisions to the Game Manager because when two fish have a collision their positions

are changed. The game manager checks this result to update the positions of fish and also to update the map.

3.5.4.1.2 Load Game

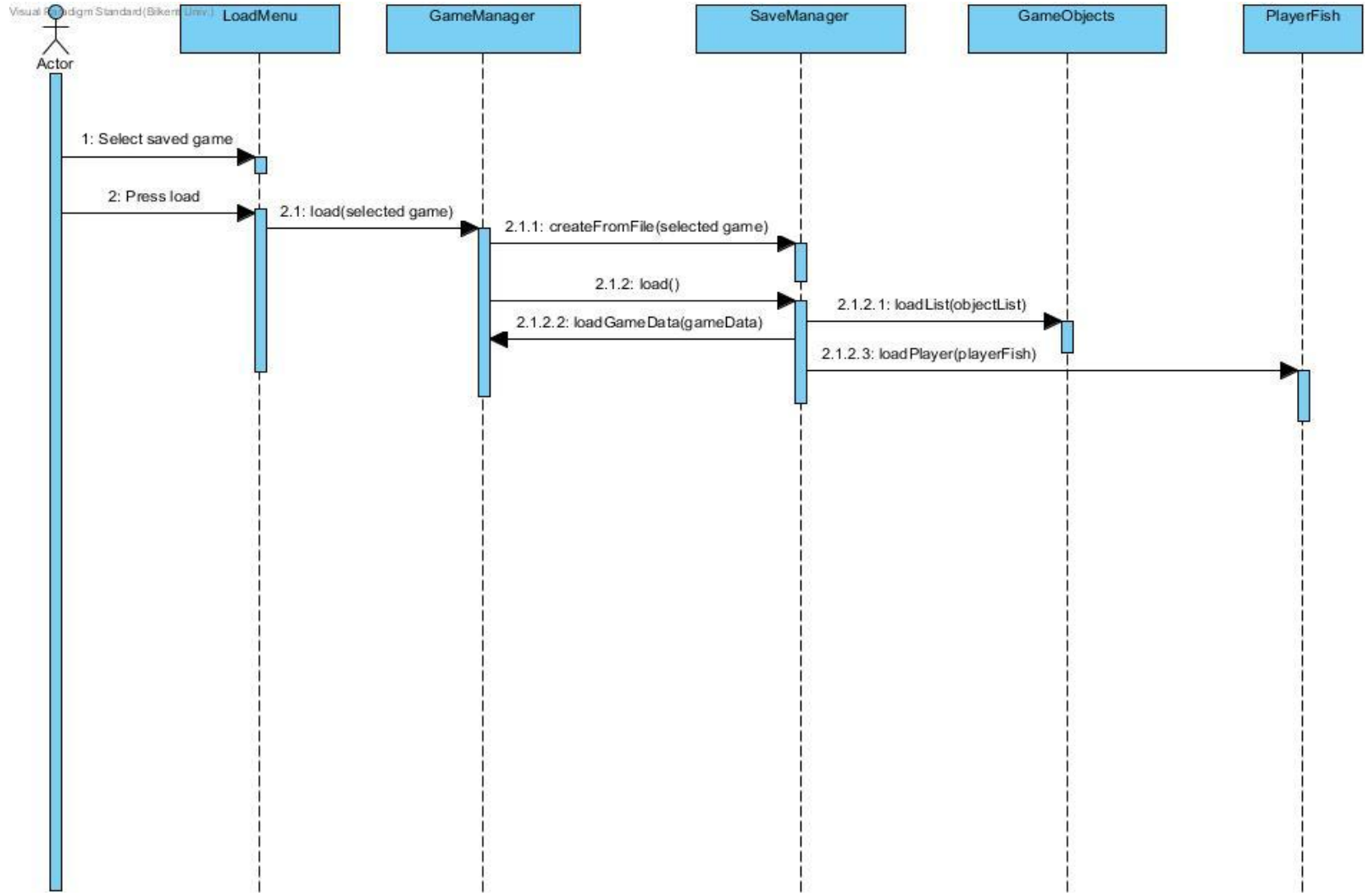


Figure 14: Load Game Sequence Diagram

When the user chooses their saved game the loading sequence is triggered. The game loads the selected game file by created files from the save manager. The game objects are taken back from the file as well as the player's fish's saved information.

3.5.4.2 Activity Diagram

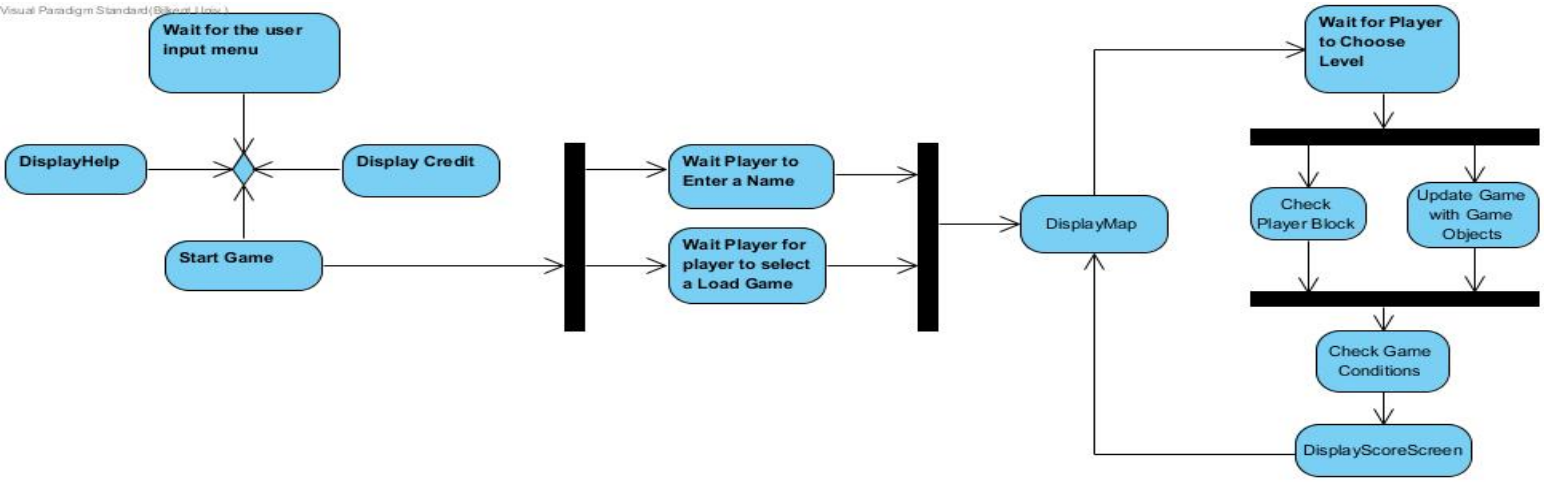


Figure 15: Activity Diagram

3.5.5 User Interface

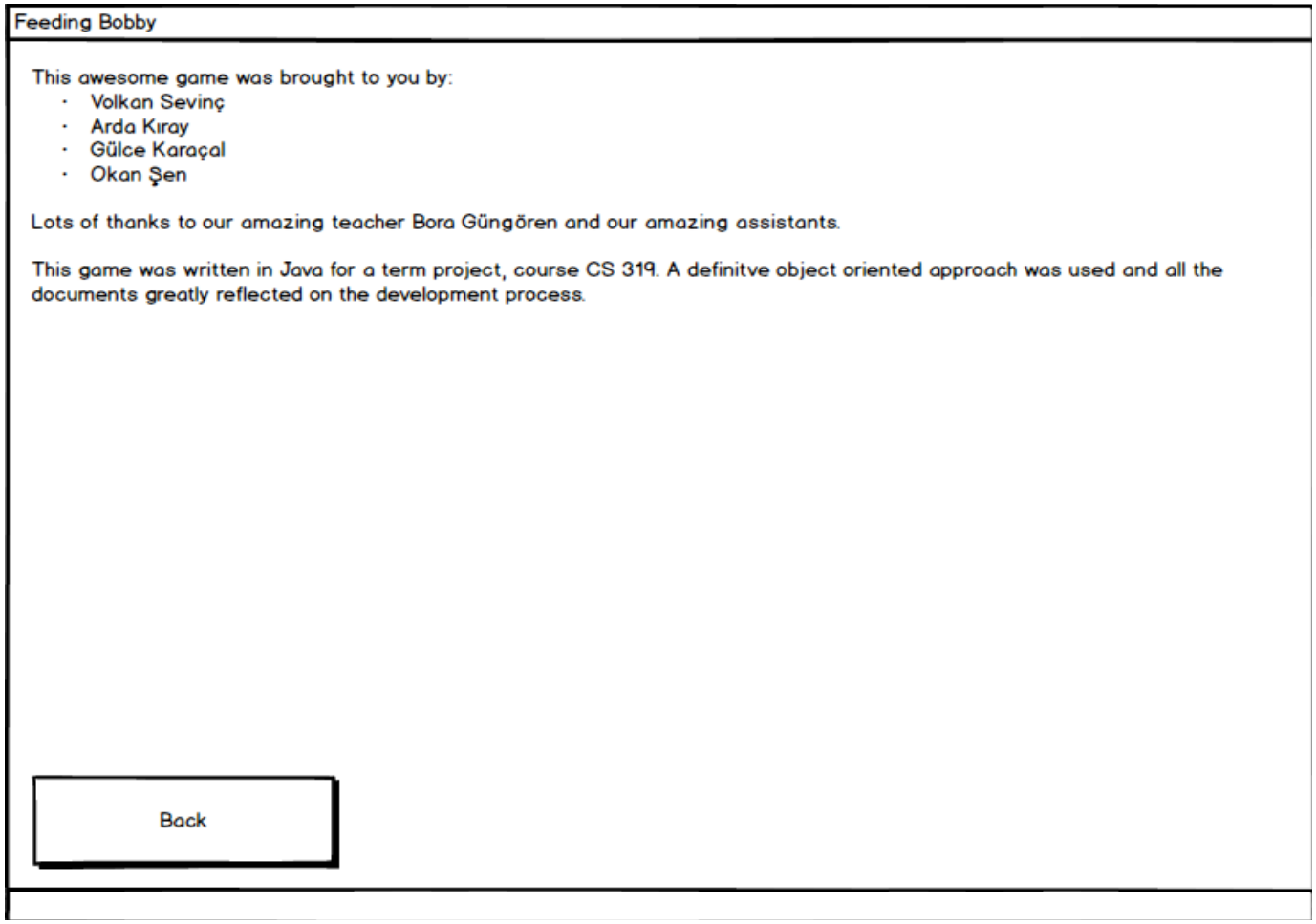


Figure 16: Credits

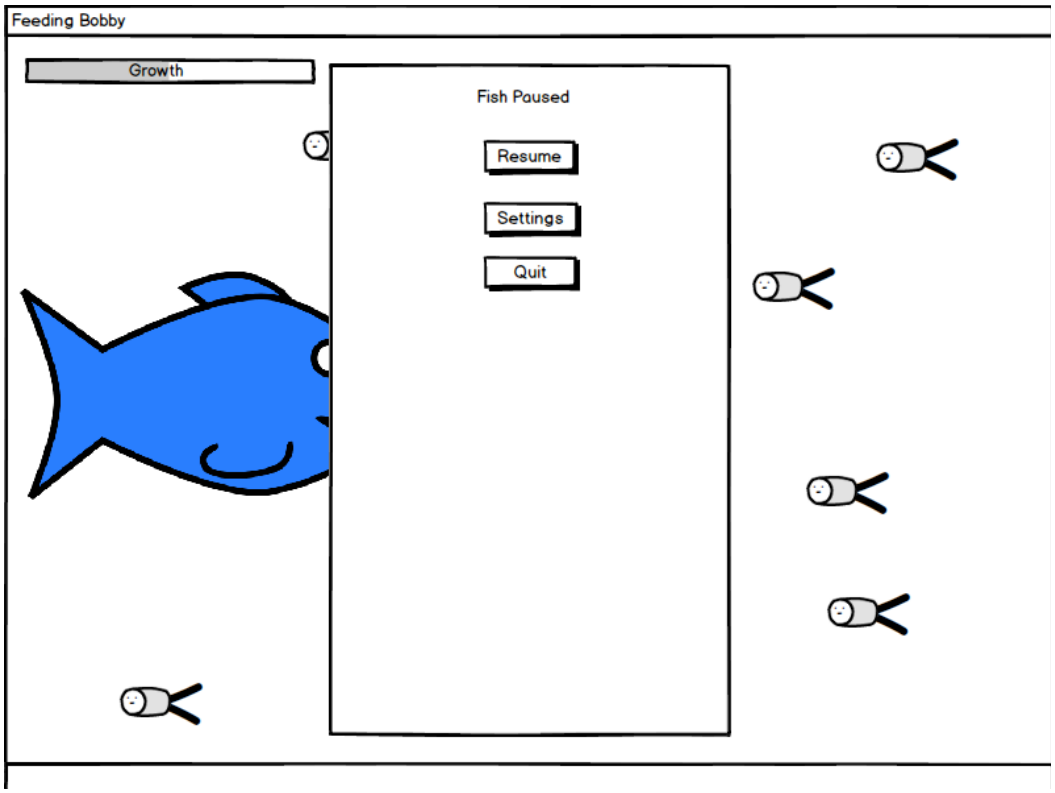


Figure 17: Pause Game

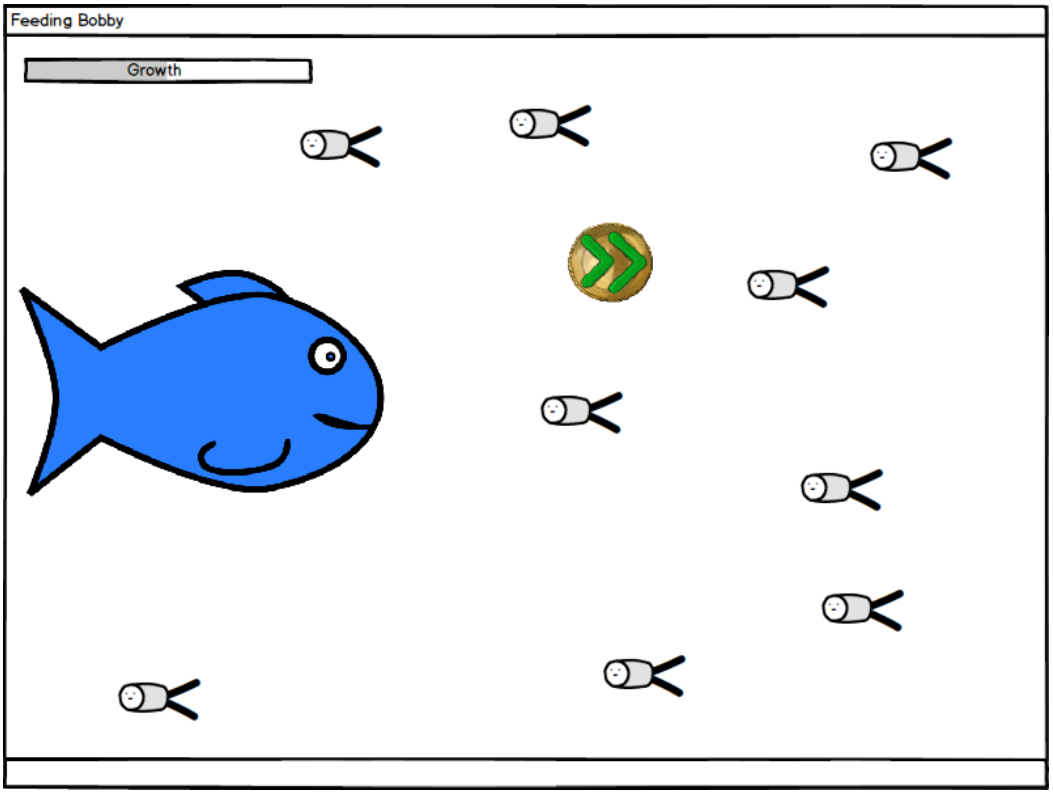


Figure 18: Game Screen

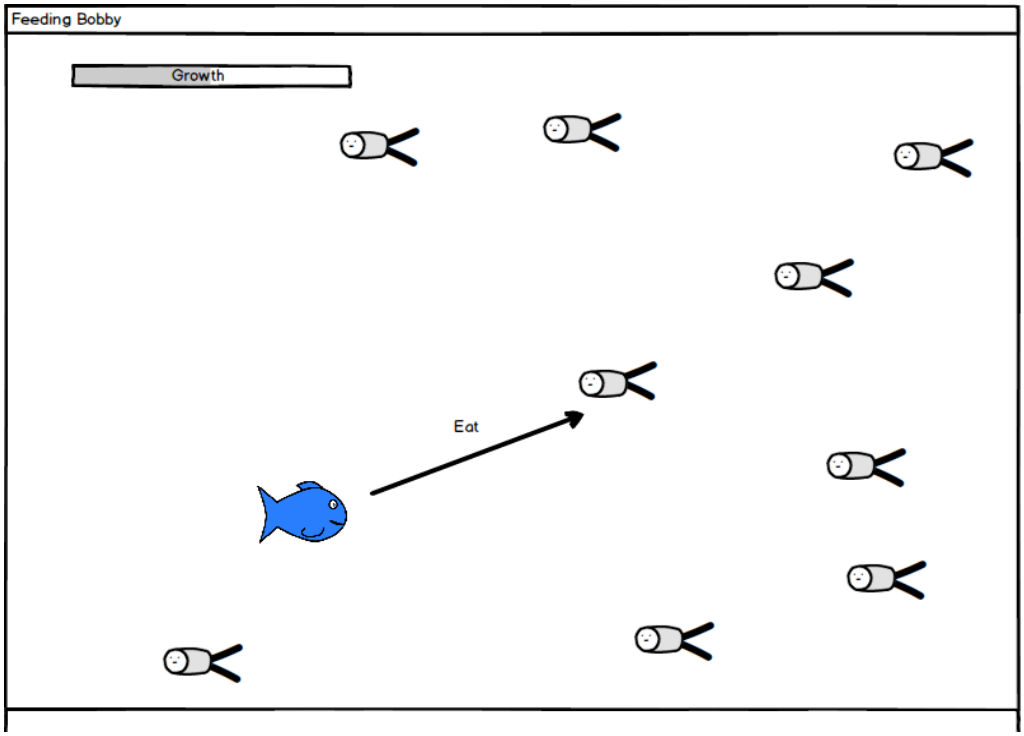


Figure 19: Before Eating

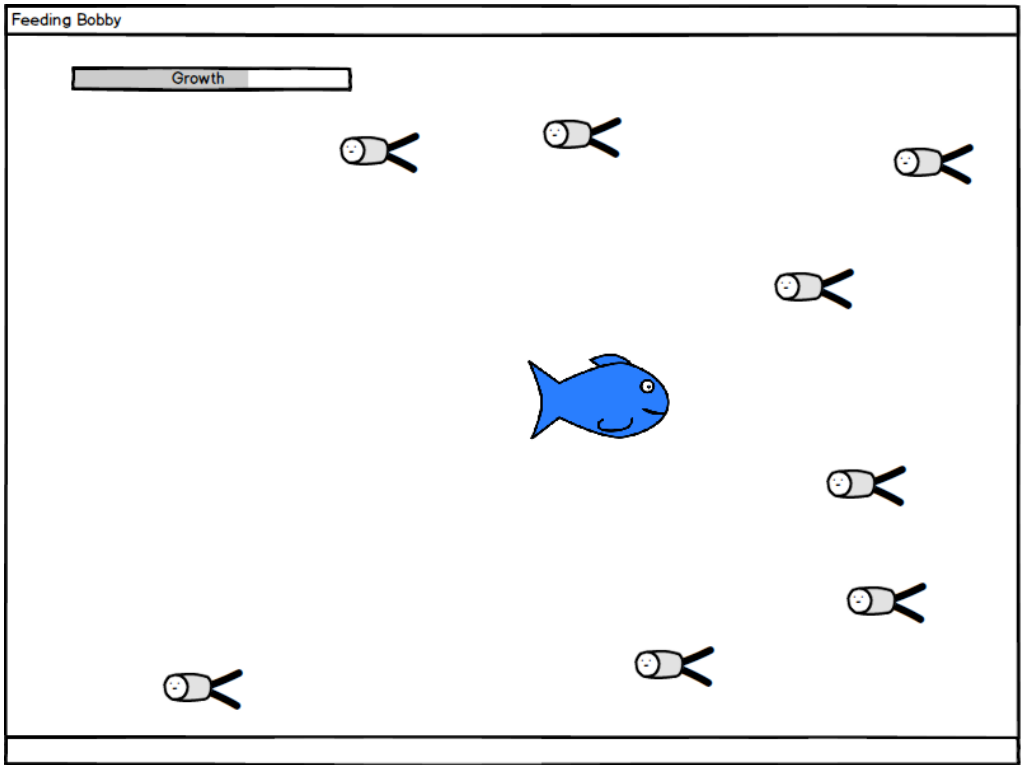


Figure 20: After Eating

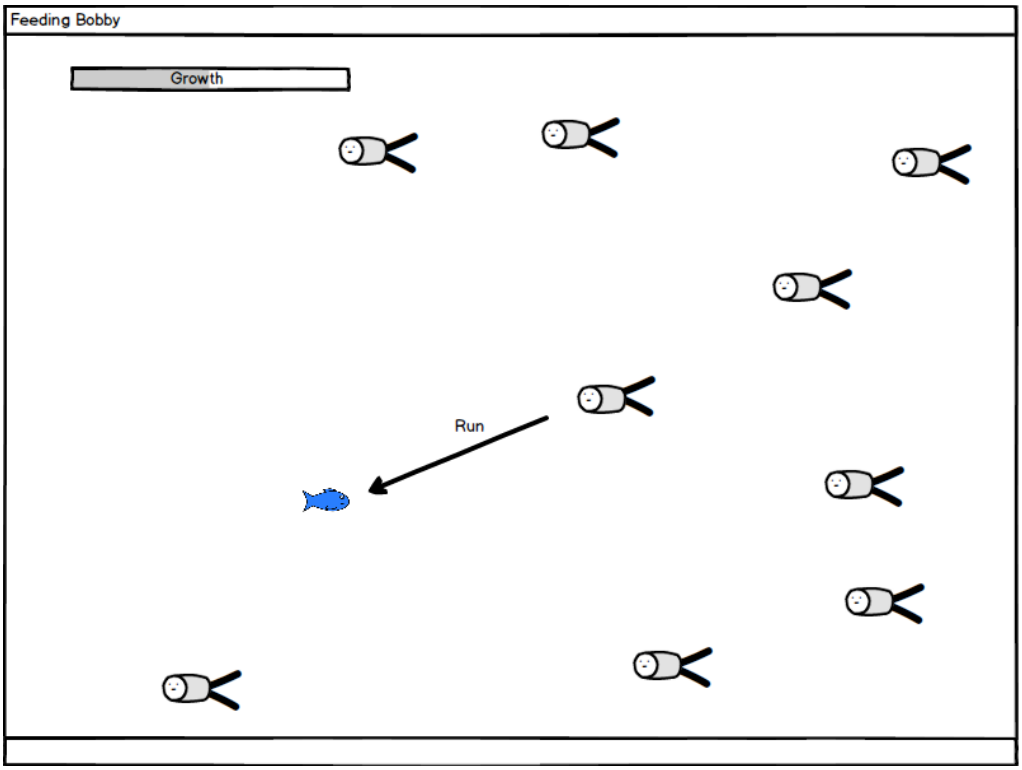


Figure 21: Escape

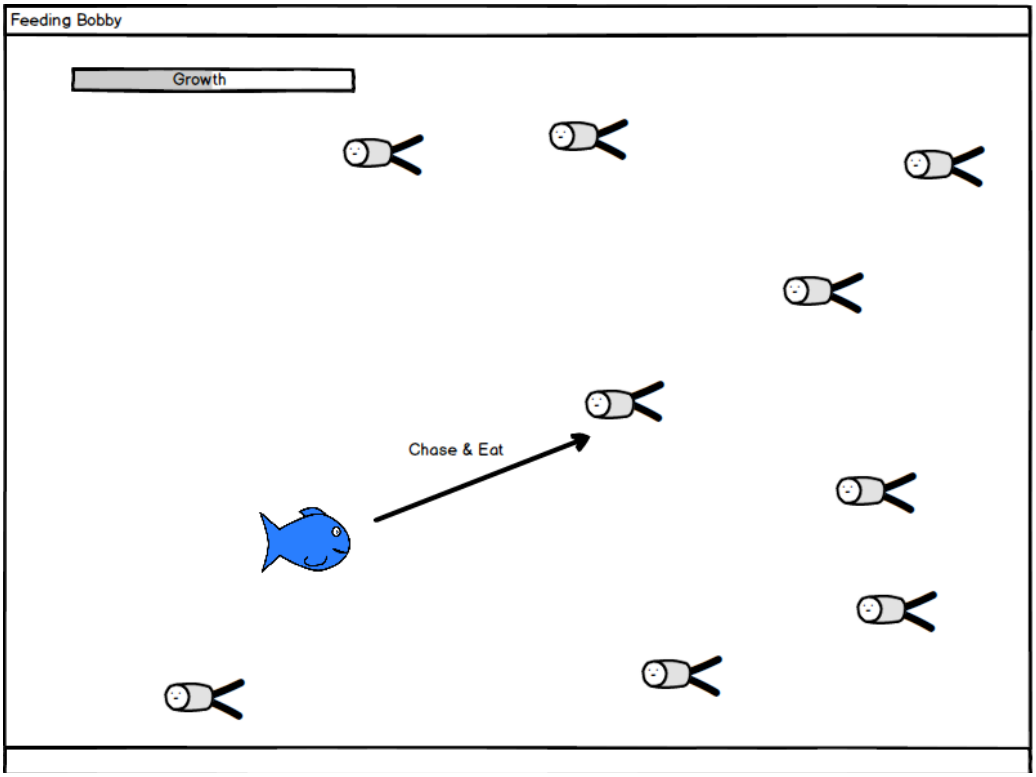


Figure 22: Chase

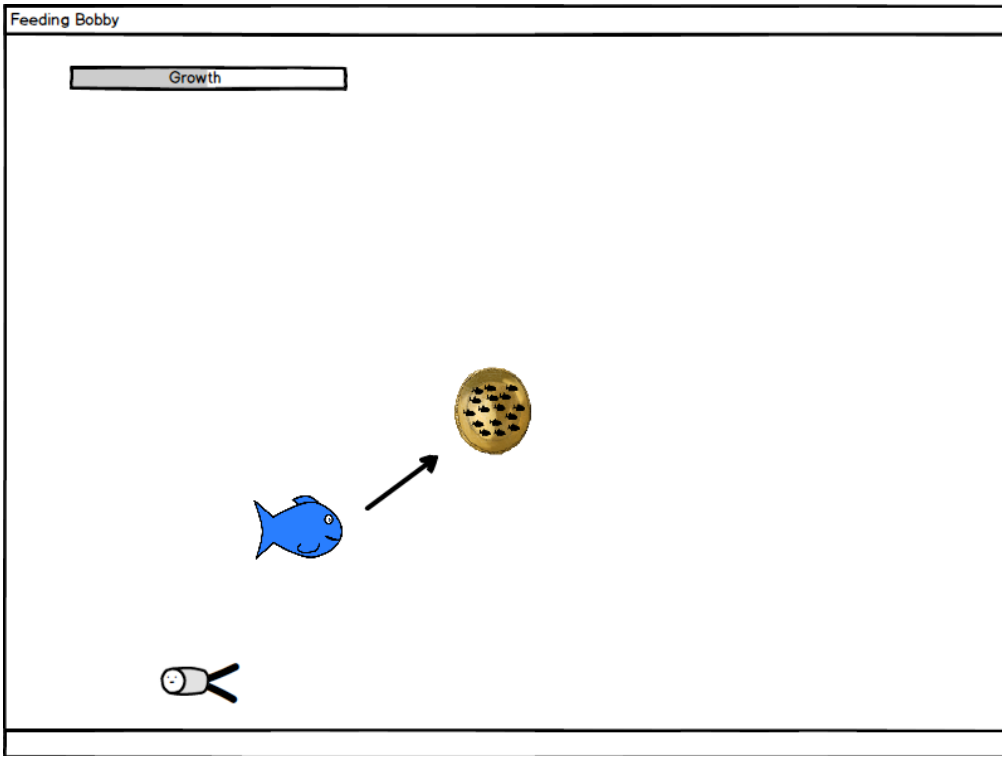


Figure 23: Before Swarm

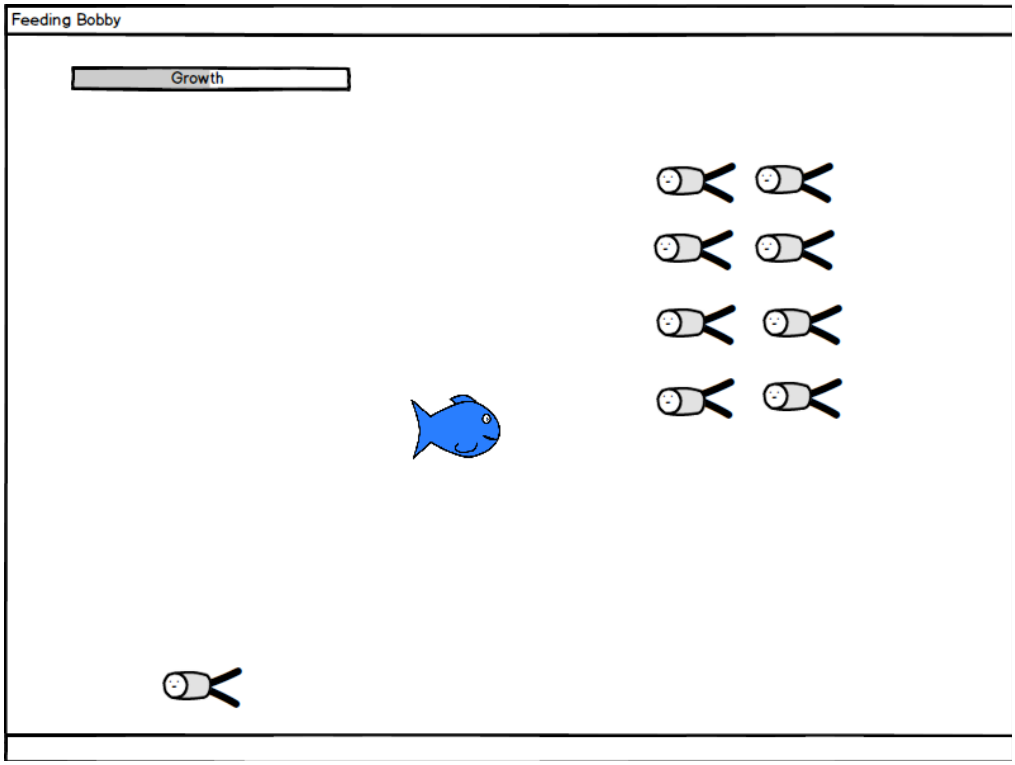


Figure 24: After Swarm

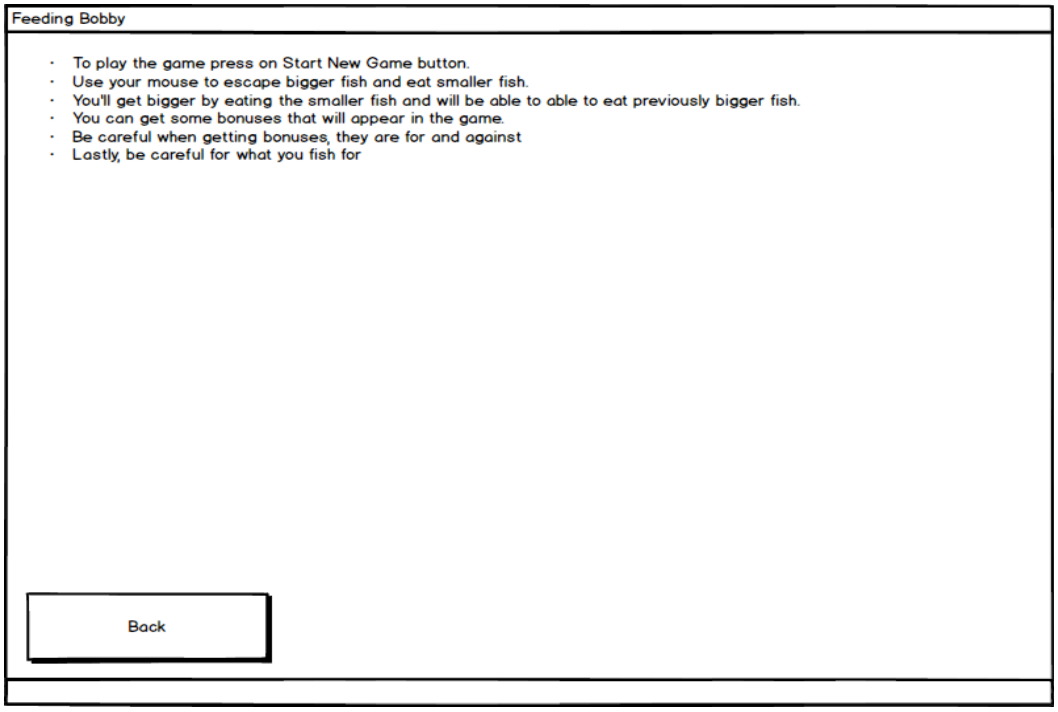


Figure 25: Help Screen

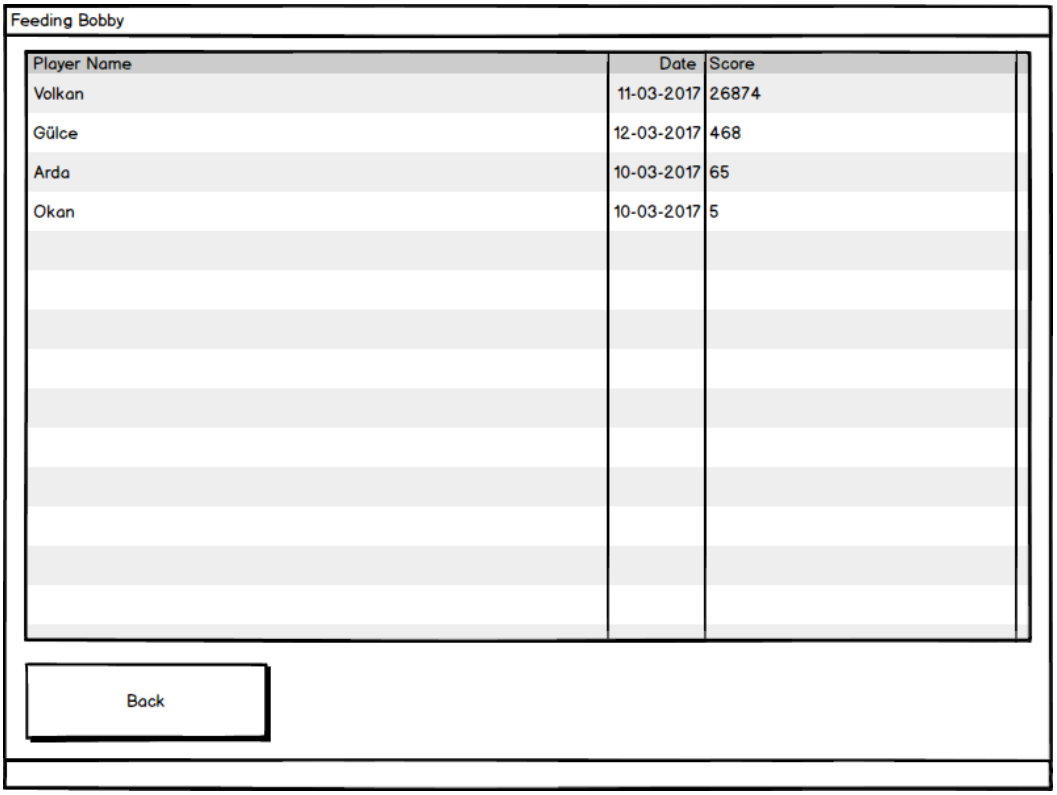


Figure 26: High Scores

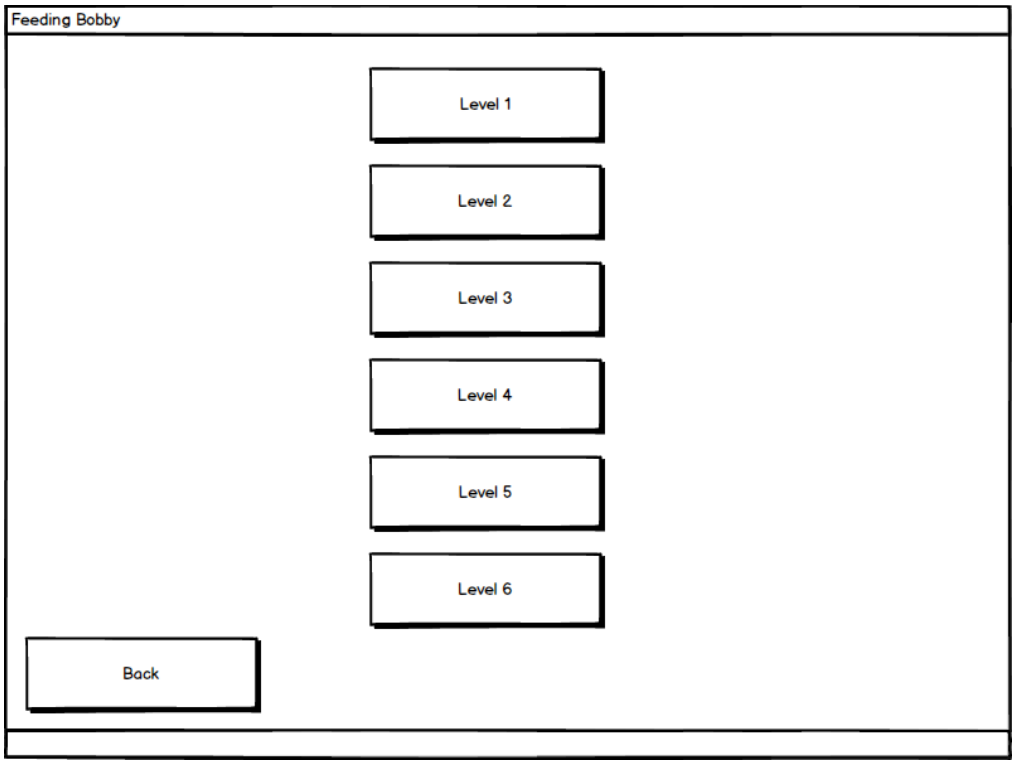


Figure 27: Journey Screen

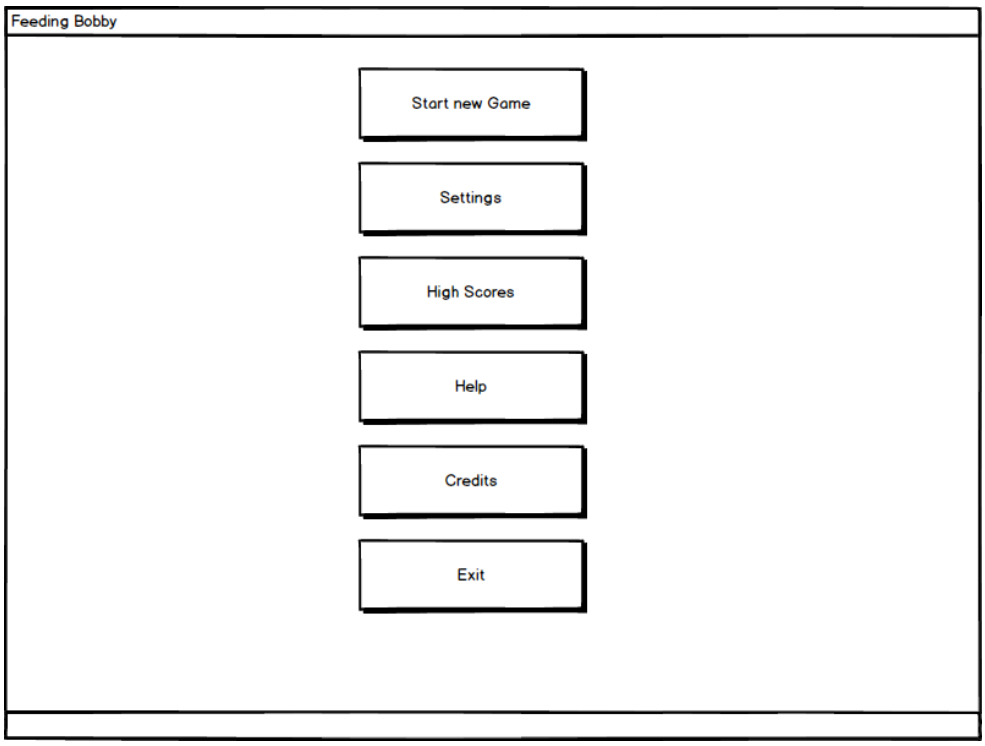


Figure 28: Main Screen

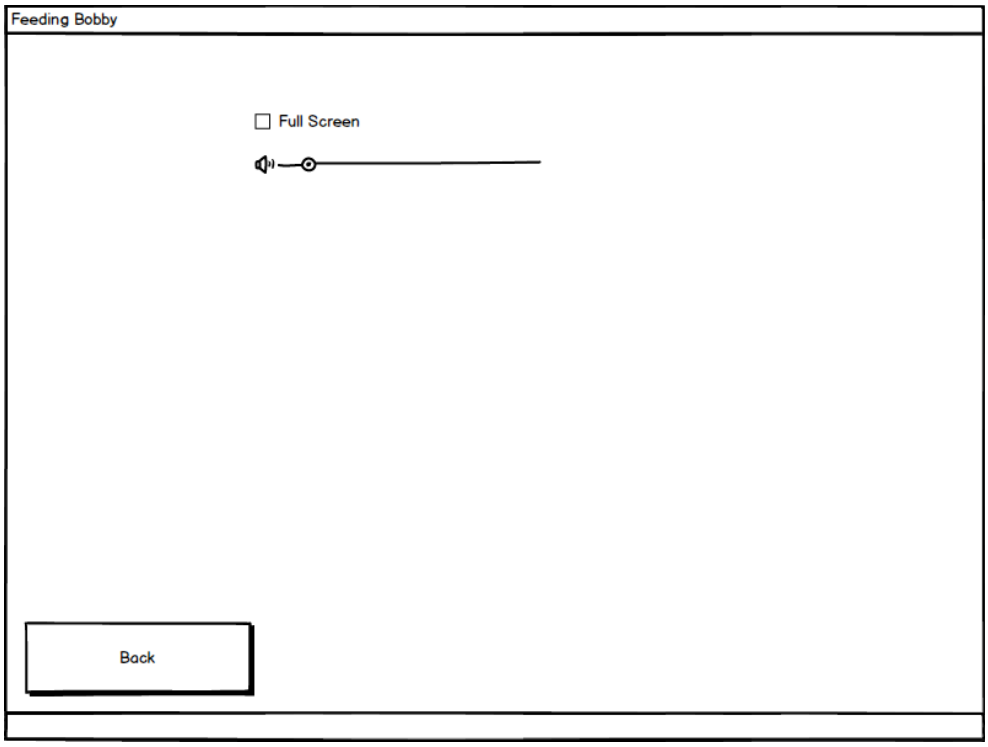


Figure 29: Settings

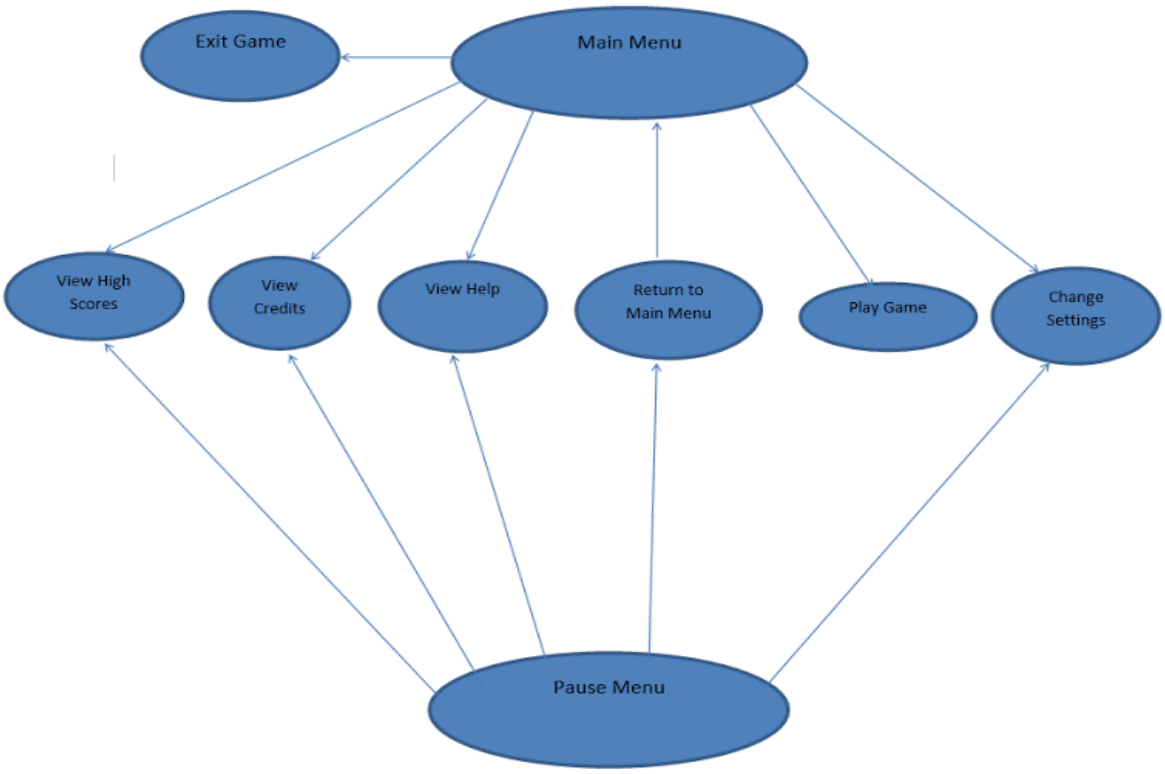


Figure 30: Navigational Path

4. Design

4.1. Introduction to Design

4.1.1. Purpose of the System

Feeding Bobby is expected to be a “user-friendly” single-player game for computers which aims to entertain users through some progressive challenges. The user will start off as a little fish and throughout the span of the game, the main goal of the user will be to get bigger by eating smaller fish, to survive by avoiding from bigger fish and to defeat the final boss while facing some mid game bosses.

First of all, we gave our priority to the speed of the game. Our aim is to make the game as fast as possible. Players will be provided high-performance and graphically qualified game. Considering this, “Processing” library developed for Java will be used.

Furthermore, due to the fact that game screen contains many icons that can cause distraction, we have decided to design the graphical user interface as “user-friendly” to prevent complexity.

Our main objective is to design a game with high performance engine and user-friendly interface to make sure that users having maximum satisfaction from the game.

4.1.2 Design Goals

Before composing the system, identifying design goals to clarify the qualities of the system is significant. In this regard, many of our design goals inherit from non-functional requirements of our system enabled in the analysis stage. Crucial design goals of our system are described below.

4.1.2.1. Criteria

Portability: Our game will be implemented in Java which enables cross-platform virtual machine (JVM) that allows user to run the game on many operating systems. Therefore, this feature of Java makes our game portable on any operating system that contains JVM.

End User Criteria

Usability: Considering the fact that we design a game, it is supposed to enable the user quality entertainment. Therefore, the system ought to provide user-friendly interfaces for menus to avoid complications that make user to have difficulties in using the system. In this manner, the system allows the player to easily find and perform the desired operations. Moreover, our system will implement actions according to mouse input from the user, such as clicking buttons, moving paddle which make the usability of the game easier.

Ease of Learning: Due to the fact that the user does not have to be familiar with the way the game is played, or with some basic concepts about the game such as loss-win conditions and power-up features, the system will provide an instructive help document, by which the user will be easily get warmed up to the game. The logic of the game is also very simple that user can easily understand intuitively or by reading the help document.

Performance: For our system, performance is a significant design goal, and therefore Processing library of Java will be used to improve the performance. For arcade games, it is crucial to provide an immediate response to players' requests to maintain users'

interests. Considering this, while displaying animations, effects smoothly for enthusiasm, our game will almost immediately create a response to player's actions.

Maintenance Criteria

Extensibility: Adding new features to the game to maintain the excitement and interest of the player is significant. Hence, our system design will provide an environment in which adding new functionalities and entities such as new power-ups to the existing system is possible.

Modifiability: Our system is going to have a multilayered structure allowing us to modify the system easily. To accomplish this, we have decided to minimize the coupling of the subsystems as much as possible in order to avoid great effects on the system components by a modification.

4.1.2.2. Trade Offs

Usability and Ease of Learning vs. Functionality:

We have discussed that our game should be easy to learn because we want the user to not to lose his interest over the game. Considering this, our main priority is the usability of the game rather than the functionality. Since the purpose of the system is providing entertainment to users, our system will not bore the player with complicated functionalities. Instead, we will focus on the ways which make our game easy to understand such as having a simple interface and familiar instructions. In this way, users can spend their time playing the game rather than struggling to learn it.

Performance vs. Reusability:

Our system design's primary concern is not reusability but getting high performance. As mentioned, we focused on to designing a game that should be able to create an immediate response to players' requests to maintain users' interests. Integrating any of our classes to another game is not one of our plans. Hence, our system will have classes which are designed particularly for the tasks so that the code will not be created more complicated than necessary.

4.1.3. Definitions, Acronyms, and Abbreviations

Java Virtual Machine (JVM): A Java virtual machine (JVM) is an abstract computing machine that enables a computer to run a Java program. [1]

Processing (Java Library): Processing is a simple programming environment that was created to make it easier to develop visually oriented applications with an emphasis on animation and providing users with instant feedback through interaction. [2]

4.1.4. References

[1] https://en.wikipedia.org/wiki/Java_virtual_machine

[2] Fry, B and Reas, C. (2007). "Processing Overview". https://processing.org/tutorials/overview/. [Accessed: Mar 17, 2017].

4.2. Software Architecture

4.2.1. Overview

In this section, we will decompose our system into maintainable subsystems. By dividing these subsystems, our main goal is to reduce the coupling between the different subsystems, while increasing the cohesion between subsystem components. We tried to decompose our system in order to apply MVC(Model View controller) architectural style on our system.

4.2.2. Subsystem Decomposition

In this section, the system is divided into relatively independent parts to clarify how it is organized. Since the decisions we made in identifying subsystems will affect significant features of our software system like performance and extendibility; decomposition of relatively independent part is crucial in terms of meeting non-functional requirements and creating a high quality software.

In Figure-1 system is separated into three subsystems which are focusing on different cases of software system. Names of these subsystems are User Interface, Game Management and Game Entities. They are working on completely different cases and they are connected each other in a way considering any change in future.

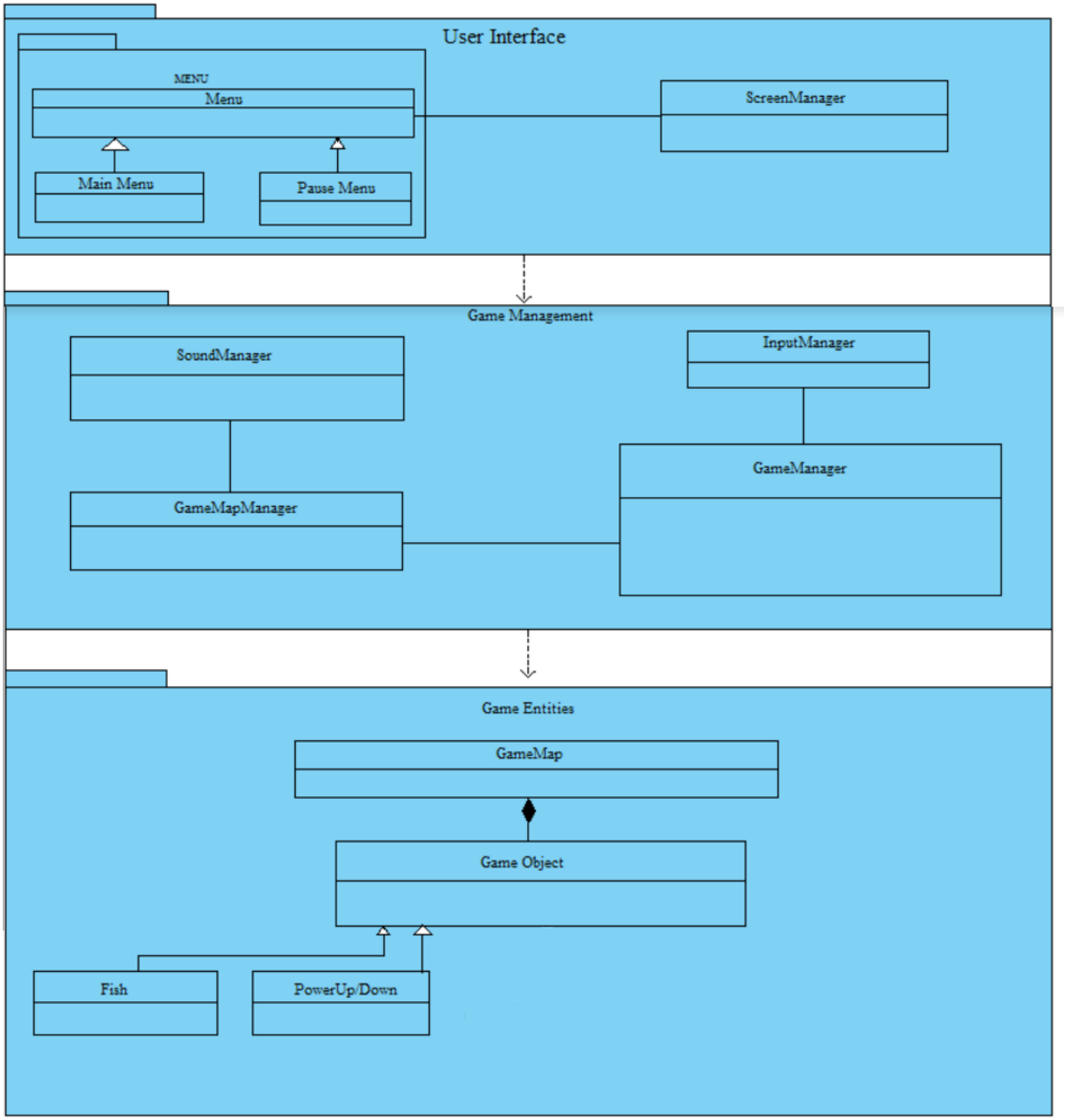


Figure 31: Basic Subsystem Decomposition

4.2.3. Architectural Styles

4.2.3.1 Layers

We decomposed our system in 3 layers: User Interface, Game Management, Game Objects. The relation between the layers is hierarchical. Since the interaction between user and the system is handled by User Interface subsystem, the top of our hierarchy is User Interface subsystem. Every interaction with user is interpreted in User Interface subsystem and the information of desired operations that needs to be performed is transmitted to Game Management subsystem. Game Management subsystem is the core part of our design. With the information of desired operations, Game Management subsystem operates on game objects. By the time, Game Objects subsystem keeps the record of game objects. It checks whether desired game object exists, changed, unchanged and keeps the track of each game object. Below is our schematic decomposition of the system.

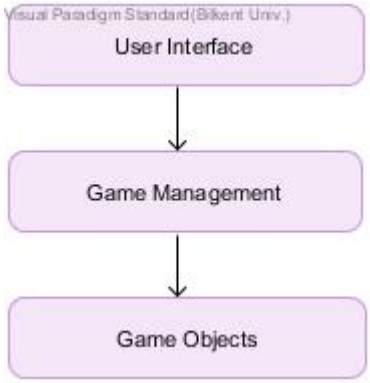


Figure 32: Layers of the System

4.2.4. Hardware / Software Mapping

Feeding Bobby will be implemented in Java programming language. As hardware configuration, Feeding Bobby needs a keyboard (for typing username) and a mouse for the user to control the titular character. Since the game will be implemented in Java, system requirements will be minimal, a basic computer with basic software installed such as operating system and java compiler to compile and run the .java file.

For storage, we will have .txt based structure to form progression data and high score list, hence the operating system should support .txt file format in order to be able to read these files. Moreover, game will not require any kind of internet connection to operate.

4.2.5. Persistent Data Management

Since Feeding Bobby does not need a complex database system. it will store progression data and high score list as text files in user storage. If the text file is corrupted, it will not cause any in-game issues regarding game objects, but system will not be able to load this corrupted data. We also plan to store sound effects, music and game objects in user storage with proper and simple sound and image formats such as .gif, .waw, .png.

4.2.6. Access Control and Security

As indicated earlier, Feeding Bobby will not require any kind of network connection. There will not be any kind of restrictions or control for access. Also Feeding Bobby will not include any user profile. Due to this , there will be no kind of security issues in Feeding Bobby.

4.2.7. Boundary Conditions

Initialization

Since Feeding Bobby does not have regular .exe or such extension, it does not require an install. The game will come with an executable .jar file.

Termination

Feeding Bobby can be terminated by clicking “Quit” button in the main menu. If player wants to quit during the game user can use the provided “Pause Menu” to return to “Main Menu” and quit. There will be no “Exit to Windows” option in in game menu. User can use “ALT + F4” shortcut during the game but it might cause some data loss depending on the last save. Since Feeding Bobby will work on full screen , there will not be “X” button at the upper right unlike windowed games.

Error

If an error occurs that game resources could not be loaded such as sound and images, the game will still start without images or sound.

5. Subsystem Services

In this section, we intended to provide detailed information about our subsystem interfaces.

5.1. Design Patterns

In our project, we use Observer Design Pattern in order to manage one-to-many dependency between objects so that whenever one object alters its state, all objects which depends on it are notified and updated automatically. Due to the fact that our game includes many fish types and power-ups, changing states in the game are supposed to be thoroughly followed. In this manner, objects can be informed about changes occurred in the game and behave according to the new states.

In this pattern, there are two kind of participants: Observer that has a function signature that can be invoked when Subject changes and Subject that maintains list of observers and also sends a notification to its observers when its state changes. Considering these, our Observer Pattern Design contains two subsystems; Game Management as observer and Game Entities as subject, in other words observable. Game Management subsystem includes some classes such as GameManager and LevelManager that will observe the game's flow of events and will inform game objects regarding the alterations in the states of other objects.

5.2. User Interface Subsystem Interface

User Interface Subsystem is used for graphical functions of our system. It also manages graphical components and Menu transactions.

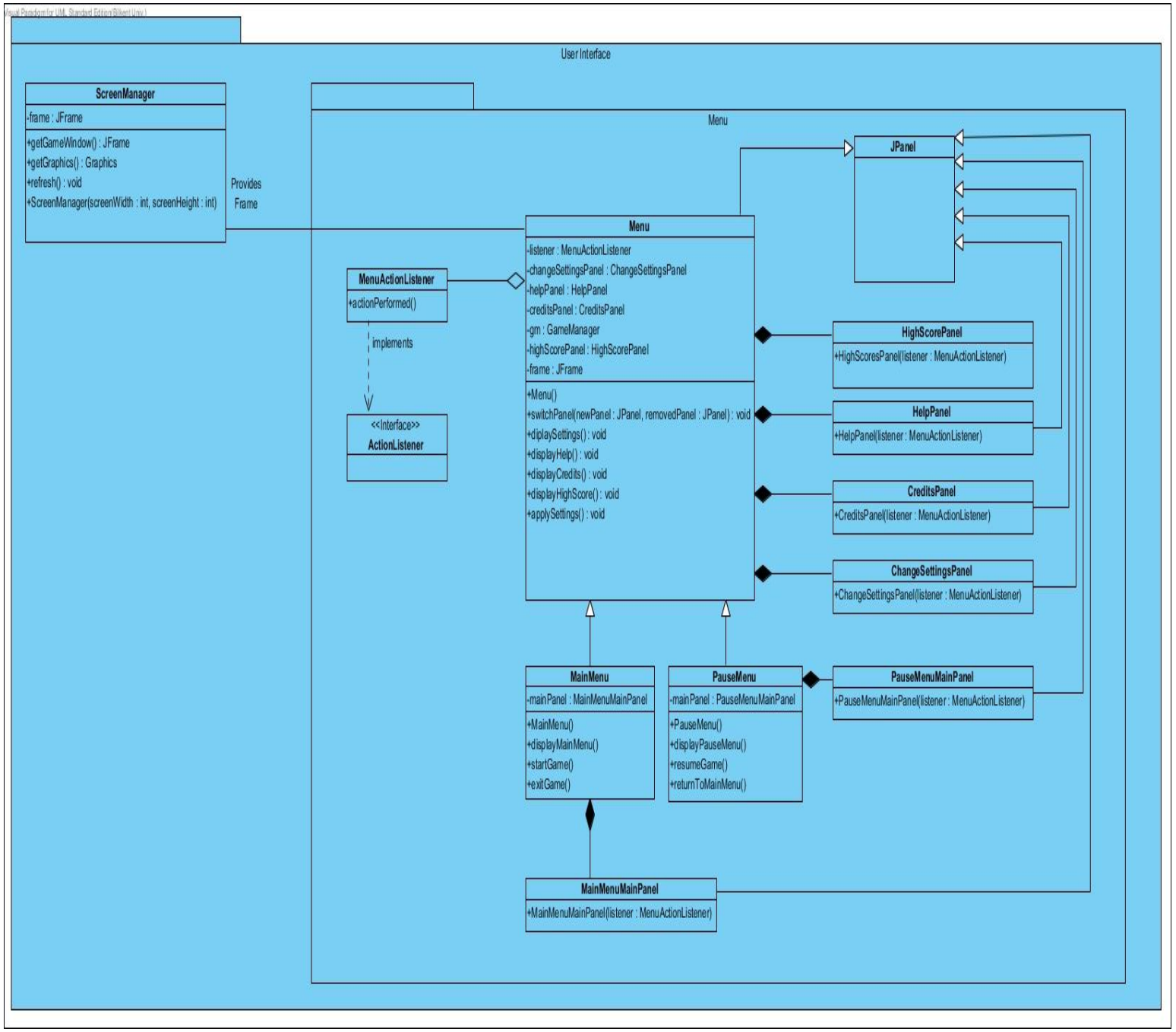
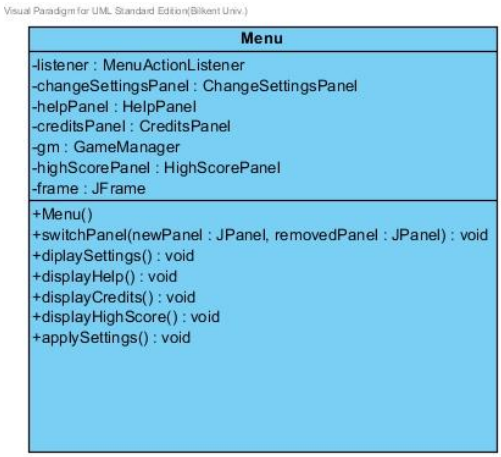


Figure 33: User Interface Subsystem Interface

Menu Class



Attributes:

- **private JFrame frame:** All visual context will be showed by using this frame.
- **private MenuActionListener listener:** This is for getting the user input from the Graphical User Interface. .
- **private ChangeSettingsPanel changeSettingsPanel:** This JPanel will be used in Graphical User Interface to display the Change Settings Menu on the screen.
- **private HelpPanel helpPanel:** This JPanel will be used in Graphical User Interface to show the Help Menu on the screen.
- **private CreditsPanel creditsPanel:** This JPanelwill be used in Graphical User Interface to show the Credits on the screen.
- **private GameManager gm:** This GameManager type property of Menu class provides reference to Game Management subsystem, when the user selects the option to Play Game.
- **private Setting settings:** This Setting type property of Menu class manages the adjustment of the system settings which are preferred by the user such as sound modifications.

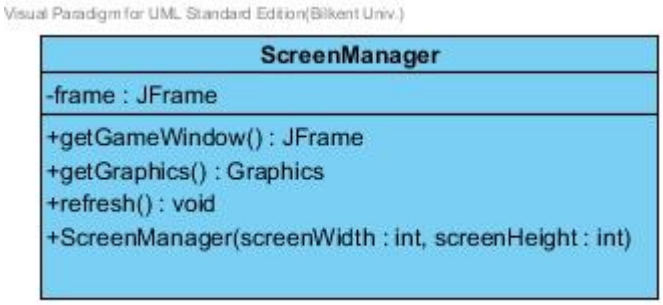
Constructors:

- **public Menu:** Initializes changeSettingsPanel, creditsPanel, helppanel, gm, settings and listener properties.

Methods:

- **public void switchPanel(newPanel, removedPanel):** It changes the panel on frame according to the option that user selects in the game menu.
- **public void displayHelp():** It contains switchPanel() method and adds *helpPanel* to frame by replacing the current panel on frame.
- **public void displayCredits():** This method includes switchPanel() method and adds *creditsPanel* to frame by replacing the current panel on frame.
- **public void displaySettings():** This method includes switchPanel() method and adds *changeSettingsPanel* to frame by replacing the current panel on frame.
- **public void applySettings():** This method applies the requested settings.

ScreenManager Class



Attributes:

- **private JFrame frame:** It is the main frame of programin which all context is displayed.

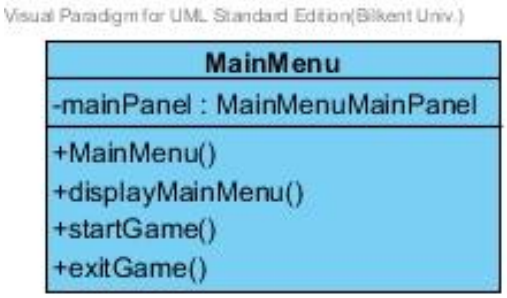
Constructors

- **public ScreenManager(int screenWidth, int screenHeight):** It takes width and height of screen as parameter to construct an object.

Methods:

- **public JFrame getGameWindow():** returns a JFrame object to display game screen.
- **public Graphics getGraphics():** returns a graphics context for drawing to an off-screen image.
- **public void refresh():** It repaints the components in the game frame.

MainMenu Class



Attributes:

- **private MainMenuMainPanel mainPanel:** This JPanel will be used in Graphical User Interface to display the Main Menu on the screen.

Constructors:

- **public MainMenu():** It initializes instances of MainMenu object.

Methods:

- **public void displayMainMenu():** It contains switchPanel() method and adds mainPanel to frame by replacing the current panel on frame.
- **public void startGame():** Through the reference of gm attribute of MainMenu class, this method invokes gameManagement subsystem to control gameplay

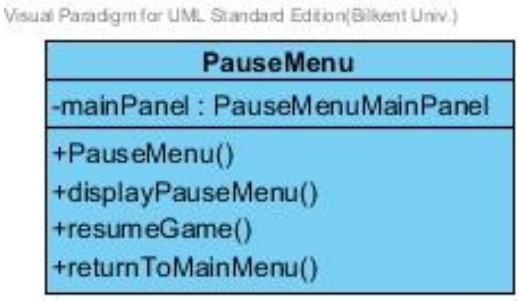
routine.

- **public void exitGame():** This method ends the run of game.

This class will be the child class of Menu Class and according to inheritance relationship between them, the Main Menu Class will inherit some methods from its parent class such as:

- displaySettings()
- displayCredits()
- displayHighScore()
- displayHelp()

PauseMenu Class



Attributes:

- **private PauseMenuMainPanel mainPanel:** This JPanel will be used in Graphical User Interface to display the Pause Menu on the screen.

Constructors:

- **public pauseMenu():** It initializes instances of PauseMenu object.

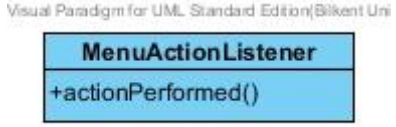
Methods:

- **public void resumeGame():** This method removes PauseMenu panel and continuous game routine.
- **public void returnToMainMenu():** This method creates a new mainMenu object and displays it on screen.

This class will be the child class of Menu Class and according to inheritance relationship between them, the Pause Menu Class will inherit some methods from its parent class such as:

- displaySettings()
- displayCredits()
- displayHighScore()
- displayHelp()

MenuActionListener



Methods:

- **public void actionPerformed(ActionEvent e):** This method overrides actionPerformed method of ActionListener interface.

HighScorePanel, HelpPanel, CreditsPanel, ChangeSettingsPanel, MainMenuMainPanel, PauseMenuMainPanel classes instantiate requested panels by the user. These are placed on frame by MenuActionListener.

5.3. Game Management Subsystem Interface

Game Management Subsystem is the one that manages game logic and game dynamics with 6 components. These components can be classified as controller classes such as GameManager, AudioManager, GameMapManager and InputManager and property classes such as Setting and GameInformation.

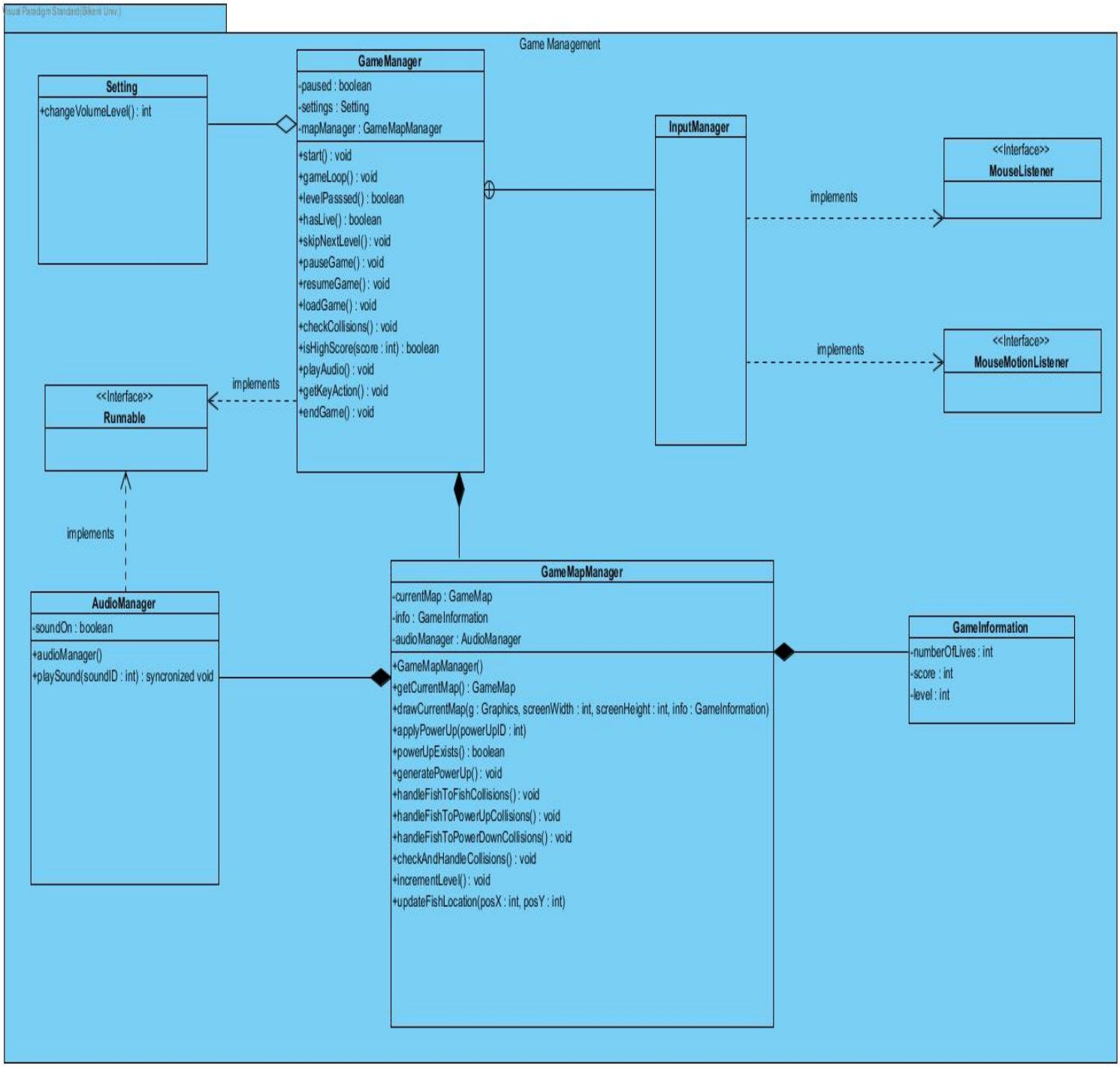
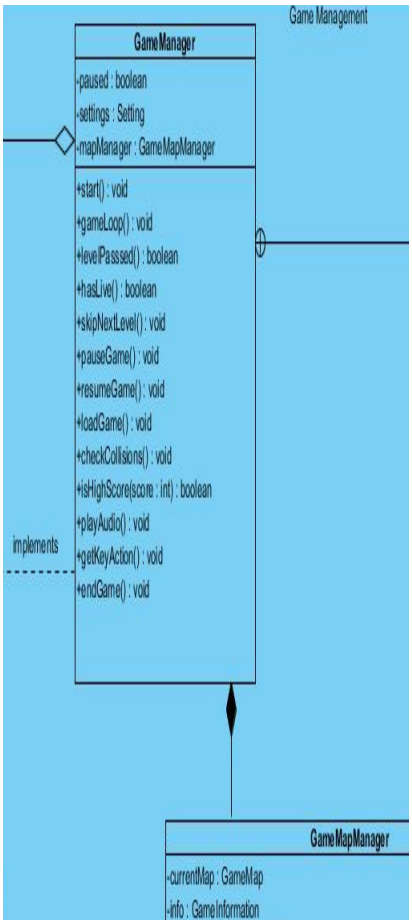


Figure 34: Game Management Subsystem Interface

Game Manager Class



- This class is the Observer Class in our design. It observes objects in the game and informs other objects whenever there is an alteration in states. This class also implements runnable interface, since the game loop runs as a thread.

Attributes:

- **private boolean paused**: It is used to check whether or not the game is paused in the game loop.
- **private Setting settings:** It holds the settings of the game such as the adjustment of sound.

- **private GameMapManager mapManager:** It is a GameMapManager object, by which GameManager class associates with proper methods of GameMapManager class.

Constructors:

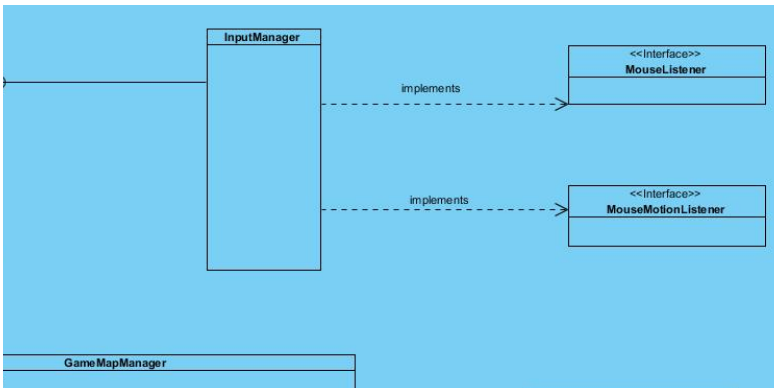
- **public GameManager():** It initializes the attributes of the GameManager for the first run of the game.

Methods:

- **public void startGame():** It starts a new game through resetting game information stored on GameMapManager class.
- **public void gameLoop():** It runs a loop in which the system is updated continuously.
- **public boolean levelPassed():** It communicates with GameMapManager class to detect whether or not the level is passed.
- **public boolean hasLive():** It checks whether or not the number of lives is bigger than zero by communicating with the GameMapManager.
- **public void skipNextLevel():** It checks whether or not the level is passed, whenever the level is passed, it increases the level number stored in the GameMapManager class.
- **public void pauseGame():** It prevents the game loop to iterate through setting paused attribute to false.
- **public void resumeGame():** It makes the game loop to keep iterating through setting the paused attribute to true.
- **public void loadGame():** It loads the game loop which had already been used.
- **public void checkCollisions():** It checks whether or not there is a collision in the game loop.
- **public boolean isHighScore(score: int):** It returns true if the given score is suitable for the high score list else it returns false.
- **public void playAudio():** It plays the sounds of the game.
- **public void getKeyAction():** It gets the action of the user by communicating with the InputManager.

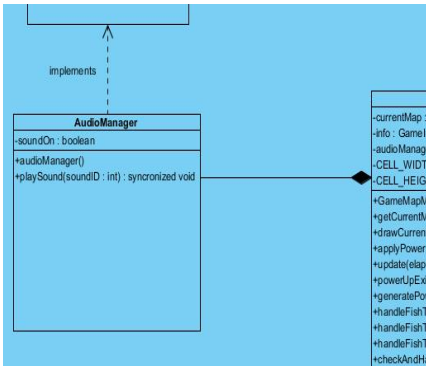
- **public void endGame():** Whenever the levelPassed method and the hasLive method both return true, this method invokes the isHighScore method.

InputManager Class



- This class is the one which determines the user actions performed by mouse. Therefore, this class implements proper interfaces of Java.

AudioManager Class



- GameMapManager uses this class whenever sounds are neededd in the game. Moreover, this class implements the Runnable Interface because AudioManager runs in a different thread.

Attributes:

- **private boolean soundOn:** It detects whether the audio is provided or is isabled

in the system.

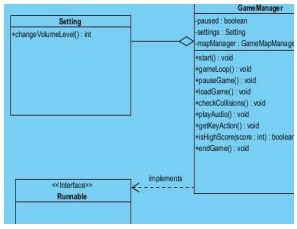
Constructors:

- **public AudioManager():** It initializes the soundOn object of this class which is set false as default.

Methods:

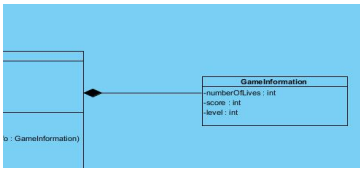
- **public synchronized void playSound(int soundID) :** GameMapManager class invokes this method when required. This method plays a audio sample according to the given value.

Setting Class



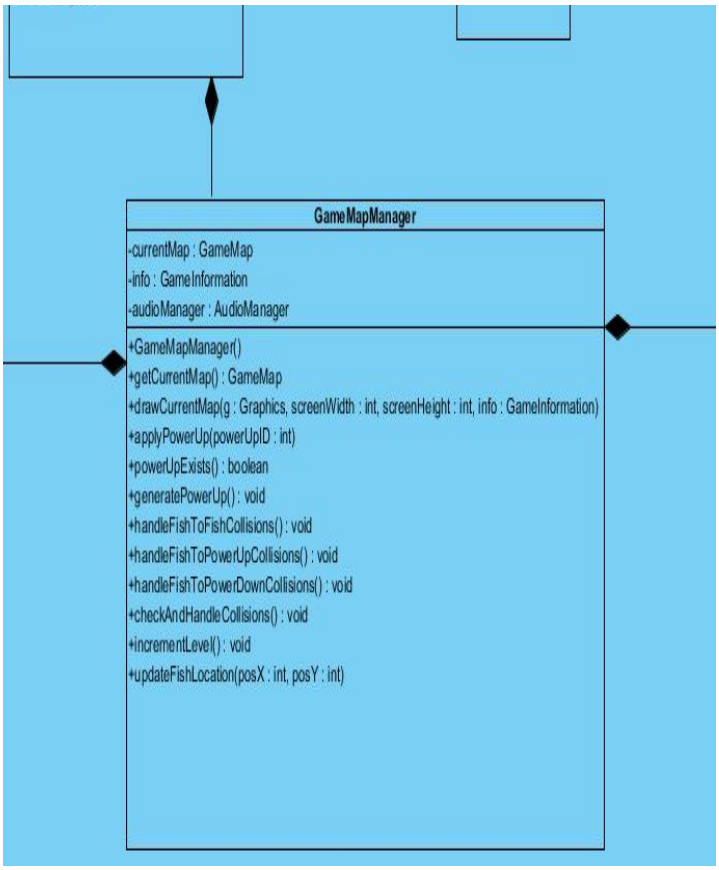
- This class is a simple property class that keeps the settings of the system. It is used by GameManager class.

GameInformation Class



- This class is a simple property class keeping the data about the current state of the game. It is used by GameMapManager class.

GameMapManager Class



Attributes:

- **private GameMap currentMap:** It performs the operations which are particular to map through referencing to this object.
- **private GameInformation info:** It holds the information about the current game such as the level number and the number of lives.
- **private AudioManager audioManager:** It references to AudioManager class to play proper sounds when required.

Constructors:

- **public GameMapManager():** It initializes a GameMapManager object with the default attribute values.

Methods:

- **public GameMap getCurrentMap():** returns the current map which is processing by GameMapManager class.
- **public drawCurrentMap(Graphics g, int screenWidth, int screenHeigh, GameInformation info):** It draws the current map according to the given attributes.
- **public void applyPowerUp(int powerUpID):** It applies the powerUp to the game through the currentMap attribute.
- **public boolean powerUpExists() :** According to a random integer, it is responsible for checking if this number is in the range of the current powerUp number. If it is in the range, method returns true.
- **public void generatePowerUp() :** It generates a random integer and generates a powerUp object with this integer by communicating with the PowerUpTable class of GameEntities subsystem, to determine which powerUp this integer corresponds to.
- **public void handleFishToFishCollisions() :** Through the reference of the currentMap object, this method finds and handles the collisions between fish objects of currentMap object.
- **public void handleFishToPowerUpCollisions() :** Through the reference of the currentMap object, this method finds and handles the collisions between fish and

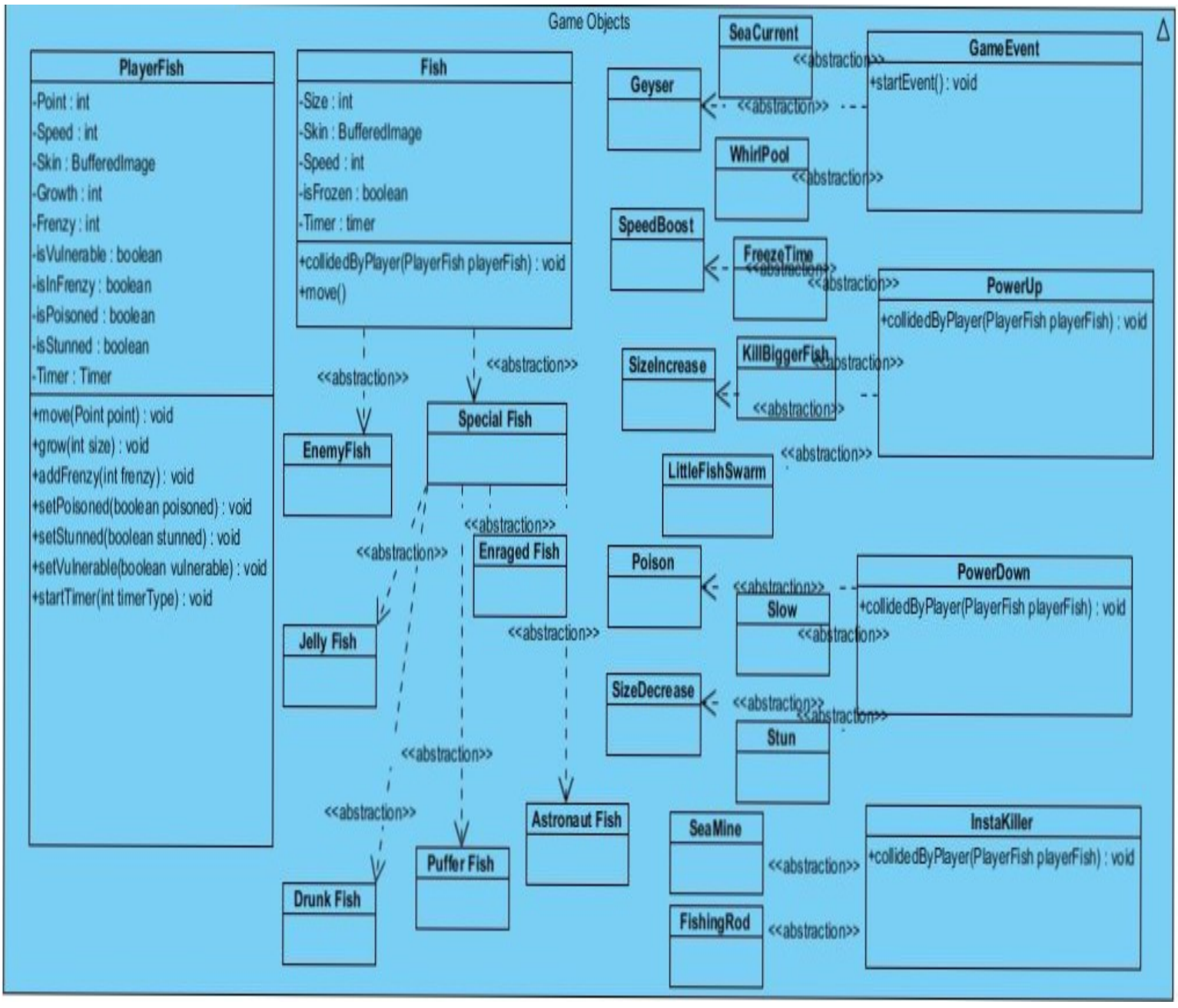
powerUp objects of currentMap object.

- **public void handleFishToPowerDownCollisions() :** Through the reference of the currentMap object, this method finds and handles the collisions between fish and power down objects of currentMap object.
- **public void checkAndHandleCollisions():** It invokes the methods of this class designed to handle the collision issues, and also finds and handles the collisions on the currentMap.
- **public void incrementLevel():** It increases the level number by one whenever the conditions of passing a level is satisfied.
- **public void updateFishLocation(int posX, int posY):** It updates the location of the fish on game map because the fish is controlled by mouse actions. Moreover, GameManager is invoked this method whenever the mouse moves.

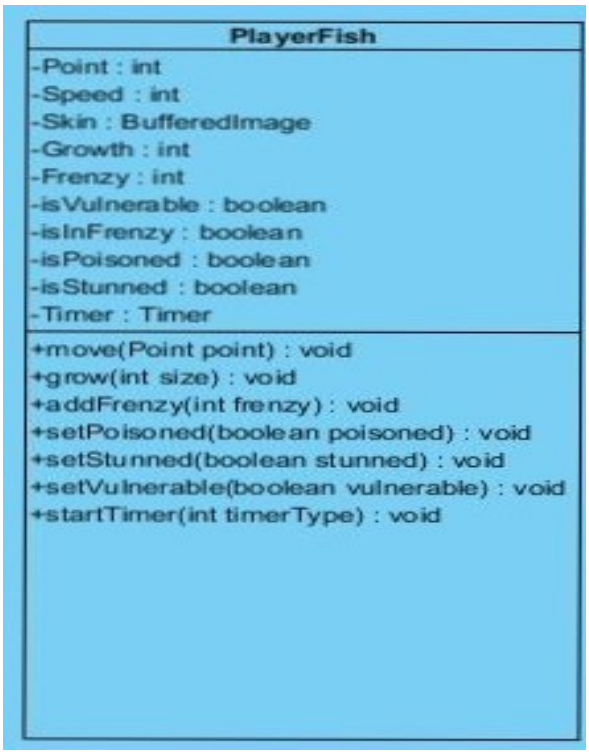
5.4 Game Objects Subsystem Interface

As the name suggests, this section deeply investigates the game objects of our system. These objects are domain-specific objects of our system. Game Object subsystem consists of 21 special game objects along with 5 abstraction classes that dictates the common behaviour of individual objects. Game Object subsystem is controlled by Game Management subsystem which is described above with details. In this section each classes of this subsystem is described in detail.

Figure 35: Game Objects Subsystem Interface



PlayerFish Class



Attributes:

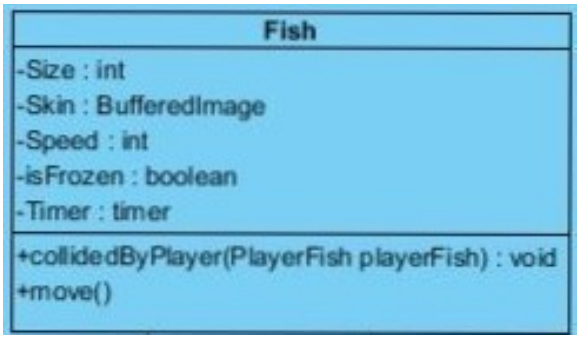
- **private int Point :** It keeps the record of the point which player's fish is at in each time. (like coordinate point)
- **private int Speed :** It is the record of player's speed that can change continuously.
- **private BufferedImage Skin :** It is a BufferedImage object which corresponds to a skin image of the player's fish.

- **private int Growth :** It keeps the current growth amount of player's fish. It has a maximum and minimum upper bounds.
- **private boolean isPoisoned :** It shows whether the player's fish frozen or not.
- **private Timer Timer :** It is a Timer which keeps the record of the cumulative elapsed time of each specific character.

Methods:

- **public void move(Point point) :** It takes one Point argument. By using given current point, move function determines the point which user desired to move.
- **public void grow(int size) :** It takes the current size as an argument and if the current growth and size are appropriate grow method does the necessary to increase the player's fish size.
- Accessor and mutator methods on the above figure basically makes the necessary changes to class members when needed. For example, if player takes the poison, setPosined function will change the isPoisoned attribute to True.
- **public void startTimer(int timerType) :** It starts the timer in order to keep the record of elapsed time of gameplay.

Fish Class



- It is a base class for various fish objects in the game. It is parent class of classes: EnemyFish, Special Fish, EnragedFish, JellyFish, PufferFish, DrunkFish.

Attributes:

- **private int Size :** It keeps the size of individual fish objects as an integer on the range of some min and max values.
- **private BufferedImage Skin :** It is a BufferedImage object that corresponds to a desired image of a skin of individual fishes.
- **private isFrozen :** It is a boolean variable which determines whether the fish is frozen or not.

Methods:

- **public collidedByPlayer(PlayerFish fish) :** It is a function which determines whether player's fish and other individual fish objects' location intersects or not. It takes PlayerFish as an argument to access its location, size and attributes to decide whether player can eat the fish or to be eaten.
- **public void move() :** It makes NPC fish objects to move randomly around the map.

EnemyFish Class

- It is a child class of Fish superclass. It is the blueprint of most basic hostile fish objects. It inherits the attributes of Fish class as well as move function of it.

SpecialFish Class

- It is the base class of all special fish objects like: EnragedFish, JellyFish, PufferFish, DrunkFish, AstronautFish. We used 2 leveled abstraction for special fishes. SpecialFish class is derived from Fish class. In that matter, SpecialFish class is a second level of abstraction for each individual special fish objects.

EnragedFish Class

- EngragedFish objects have maximum amount of speed in order to seem like enraged. It has attributes of SpecialFish class as well as additional final int Speed value and higher amount of size.

JellyFish Class

- It is another special fish class. It has the attributes and methods of SpecialFish class and a BufferedImage of jelly fish.

DrunkFish Class

- Drunk fishes' speed changes randomly to make them move like drunk person. It is also derived from SpecialFish class. It has a BufferedImage of drunk fish.

PufferFish Class

- PufferFish class is another derivation of SpecialFish class. It has a BufferedImage of a puffer fish.(with knife and hoodie). It moves directly to the Player's fish.

GameEvent Class

- GameEvent class is the base class for all random visible changes occurs on the game screen such as : Whirlpool, Geyser, SeaCurrent. Events starts randomly and ends when the user interacts with events or pass-by them.

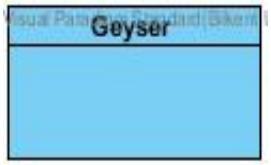
Methods:

- **public void startEvent():** It starts a random event and controlled by GameManager subsytem. This method is inherited by each specific game events.

SeaCurrent Class

- SeaCurrent class controls the random currents in the sea. It uses the GameEvent classes abstractions to start an event. Current moves every objects position in the direction of itself.

Geyser Class



- Geyser objects are like SeaCurrents. They occurs randomly and moves according objects vertically upward. It is also controlled by GameManagement subsystem.

PowerUp Class

- PowerUp Class is the base class for all power-ups such as: SpeedBoost, FreezeTime, KillBiggerFish, SizeIncrease, LittleFishSwarm. Power-ups have positive effects on user's journey. All power-ups appear randomly on the screen.

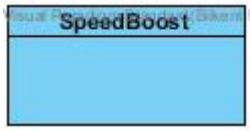
Methods:

- **public void collidedByPlayer(PlayerFish playerFish) :** It takes PlayerFish object as an argument in order to determine its position. When collision occurs power-ups considered as taken and changes are made on PlayerFish accordingly.

FreezeTime Class

- FrezeeTime Class is a subclass of PowerUp Class. It uses the same collidedByPlayer for determining the location however, it changes PlayerFishes attributes according to FreezeTime effect. It simply freezes time and objects except PlayerFish object.

SpeedBoost Class



- SpeedBoost Class is another subclass of PowerUp class. It again uses the method collidedByPlayer for determining the locational colapses and speed up the PlayerFish by changing its speed attribute to maximum.

PowerDown Class

- It is the base class for all power-downs which appears randomly on the gameplay screen. It uses the same logic of PowerUp class. Power-ups and downs use the same logic to apply the changes to PlayerFish. It is an abstraction class for classes: Posion, Slow, SizeDecrease, Stun.

Methods:

- **public void collidedByPlayer(PlayerFish fish) :** It determines whether the PlayerFish's location collapses with PowerDowns location. Collide recognition logic are used as same by all its subclasses. However, the effects to PlayerFish and surroundings differ in each subclass. It is not an operating function. It is a template for its subclasses.

Slow-Poison-Stun-SizeDecrease Classes

- Those classes are subclasses of PowerDown class. They are operating by using the same collision logic for determining whether user takes the power-down or not. Slow Class changes PlayerFish object's speed attribute to minimum. Poision Class slows down the speed of PlayerFish object and disable the eating property of PlayerFish while a collision occurs. It changes a variable notPoisoned to False in the collidedByPlayer function in order to make PlayerFish poisoned. Stun Class changes PlayerFish's speed attribute to 0 which makes him stunned for 3 seconds. SizeDecrease classes objects decrease the size attribute of PlayerFish object if encountered.

5.5. Detailed System Design

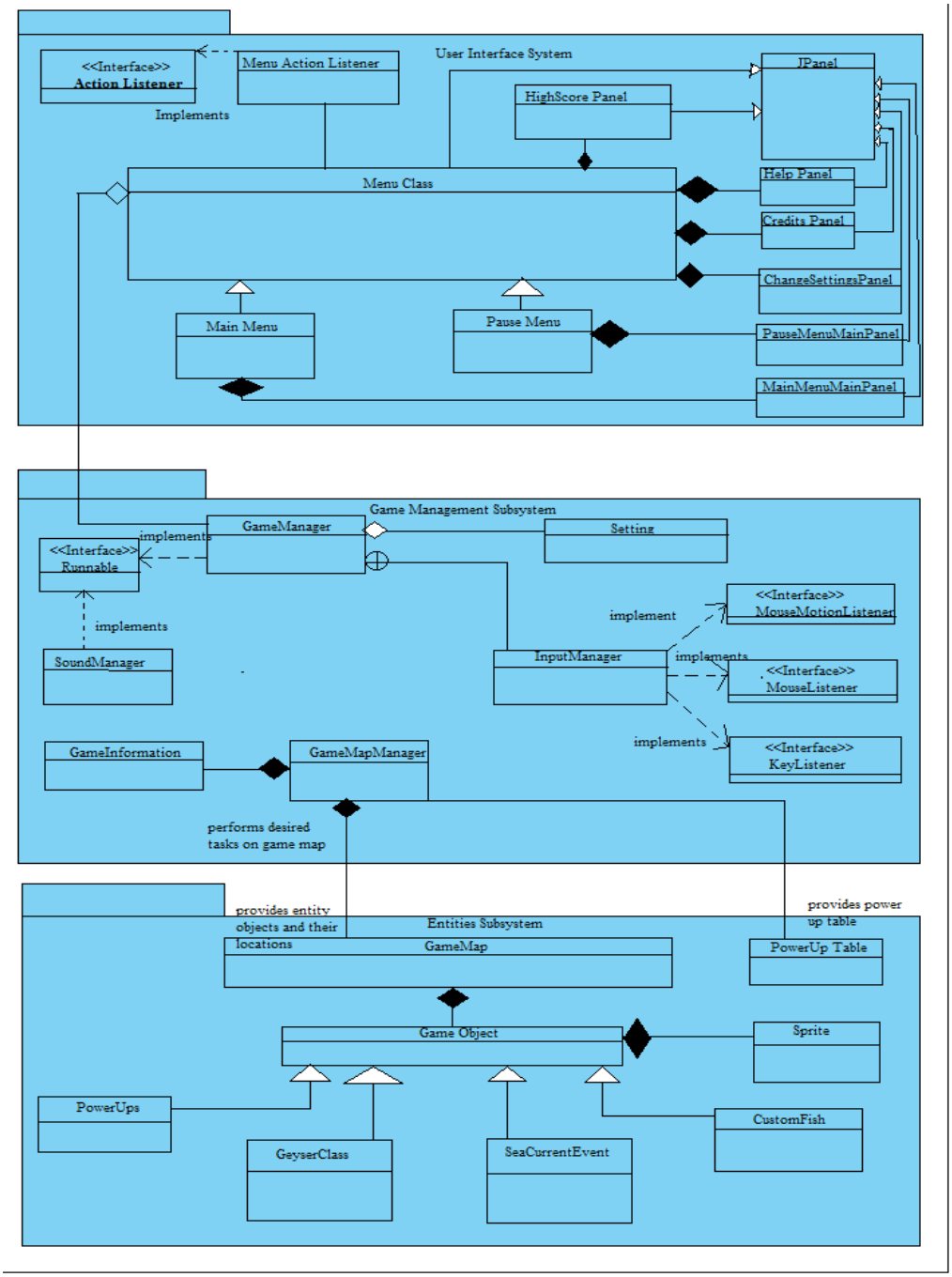


Figure 36: Detailed System Design

6. Conclusion & Lessons Learned

6.1 Conclusion

In the Analysis Report, we designed our project and provided a detailed design for “Feeding Bobby” arcade game. This report consists of two parts that are Requirement Elicitation and Analysis.

In the Requirement Elicitation part, an overview was given about our game and also, functional and non-functional requirements were provided according to the operations that the system should do and the system’s behaviors. We paid attention to consider all possible operations that the user is able to perform. Hence, we tried to find all possible situations in the game and created our scenarios considering them. While creating our models, our main goal is to fulfill these requirements and scenarios we came up with. To complete the requirement elicitation part, we created our use case model based on these scenarios and user interfaces to show the interaction between the game and the user.

In the Analysis, we provided more detailed explanation and design decisions of “Feeding Bobby” with the help of object models as well as dynamic models. For our object model, we created the class diagram of the game and for our dynamic models, we gave sequence diagrams and an activity diagram.

After the Analysis report, we started to write our design report which includes the design goals that we want to accomplish, the software architecture of the system in which the subsystem decomposition was shown and the subsystem services in which the design pattern was mentioned and the explanations of every classes with attributes and methods in each subsystem was provided.

In conclusion, by writing this report, our goal was to design and implement the game easily in the future. While writing, we paid attention to be specific and precise as much as possible to have a better understanding of our problem statement as well as

our solution domains since this report is going to be our future guide so this is the reason why, we considered the significance of this report and took it seriously.

6.2 Lessons Learned

Through the semester, we learned that the hardest parts of the software engineering is the coordination and being able to plan ahead rather than the implementation part. We learned that communication is as essential as the knowledge. We learned the importance of the designing phase and how it directly effects the final product as well as the implementation stage. We also learned how to work as a team and write reports and code while keeping the fact that people other than ourselves will try to understand in our minds. This lead us to be more organised in all our work