CS 319 – Object-Oriented Software Engineering

Project Design Report

Feeding Bobby

Group 1-K:

Arda Kıray

Gülce Karaçal

Volkan Sevinç

Okan Şen

Table of Contents

EnragedFish Class
JellyFish Class
DrunkFİsh Class
PufferFish Class
GameEvent Class
SeaCurrent Class
Geyser Class
PowerUp Class
FreezeTime Class
SpeedBoost Class
PowerDown Class
Slow-Poison-Stun-SizeDecrease Classes

3.5. Detailed System Design

1. Introduction

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.

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.

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.

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.

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]

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].

2. Software Architecture

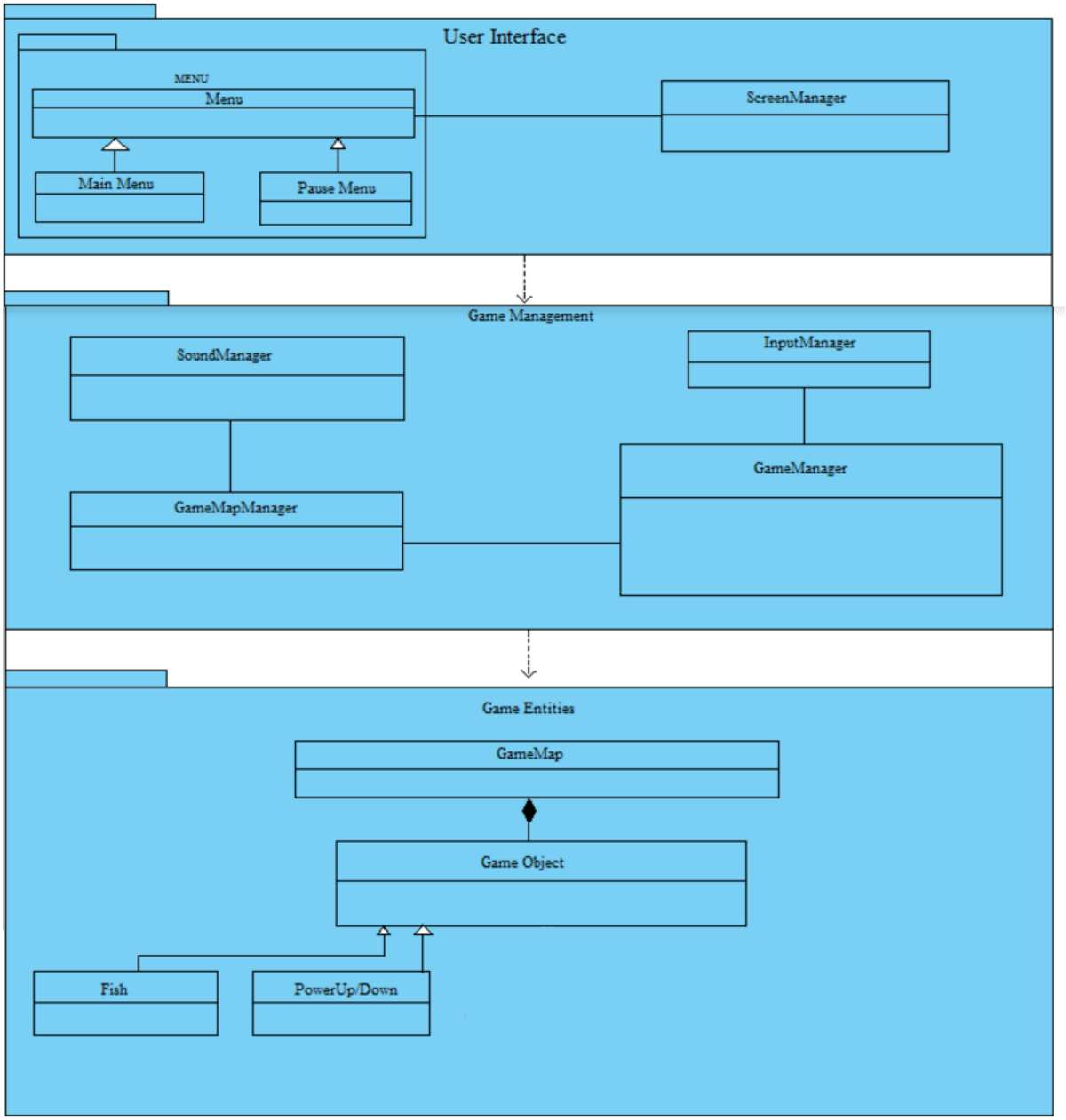
2.1. Overview

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.

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.

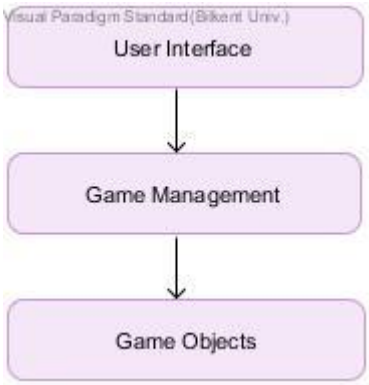


Basic Subsystem Decomposition

2.3 Architectural Styles

2.3.1 Layers

We decomposed our system in 3 layers: User Interface, Game Management, Game Objects. The relation between the layers are 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.



Layers of the System

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.

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.

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.

2.7. Boundary Conditions

Initialization

Since Feeding Bobby does not have regular .exe or such extension, it does not require an install. Though 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.

3. Subsystem Services

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

3.1. Design Patterns

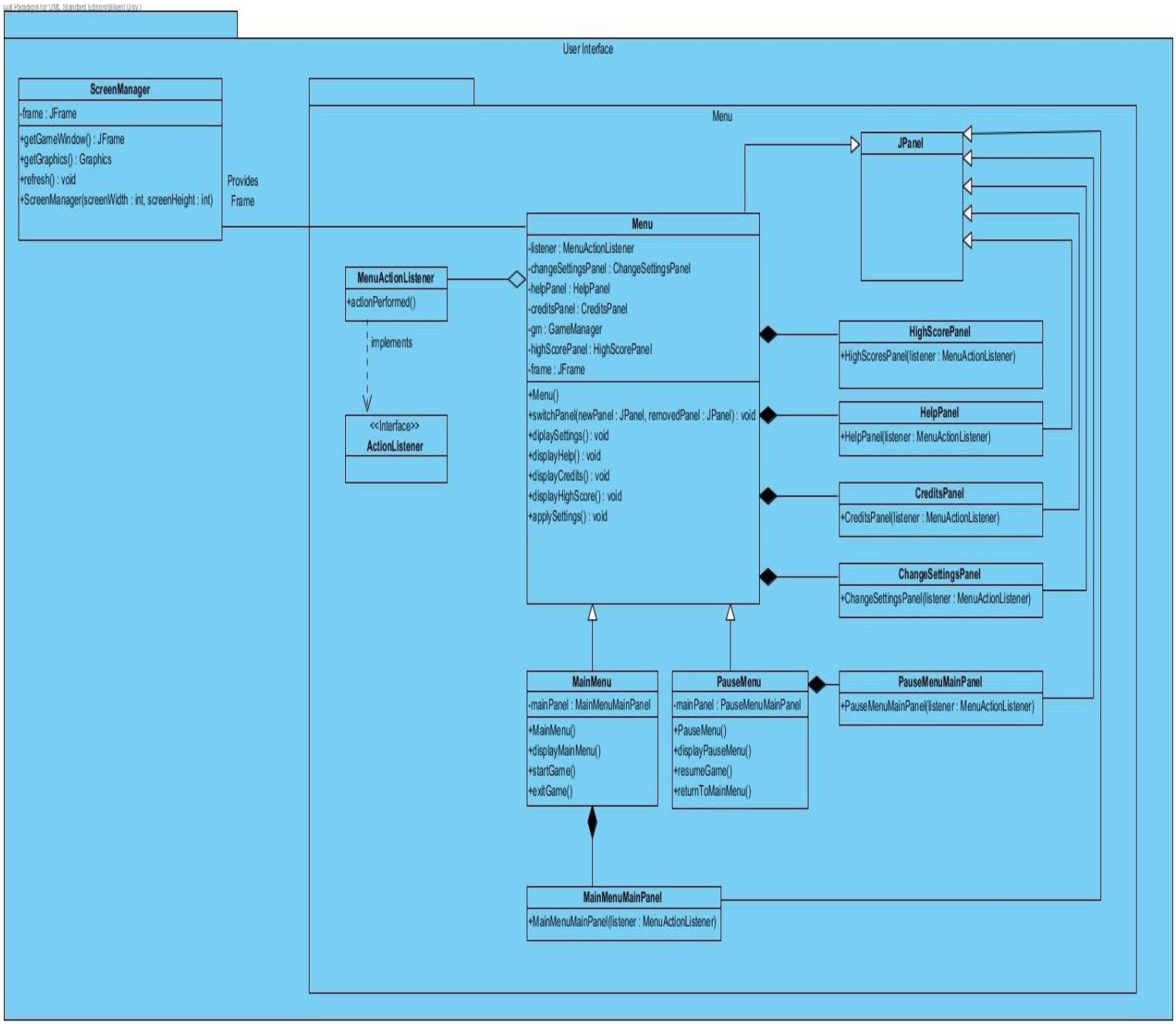
In our project, we use Observer Design Pattern in order to manage one-to-many dependency between objects so that whenever one object alters its state, all objects which depends on it are notified and updated automatically. Due to the fact that our game includes many fish types and power-ups, changing 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 subsystem: 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.

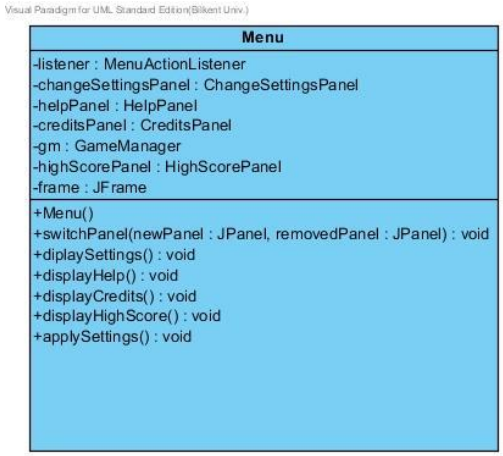
3.2. User Interface Subsystem Interface

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



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.

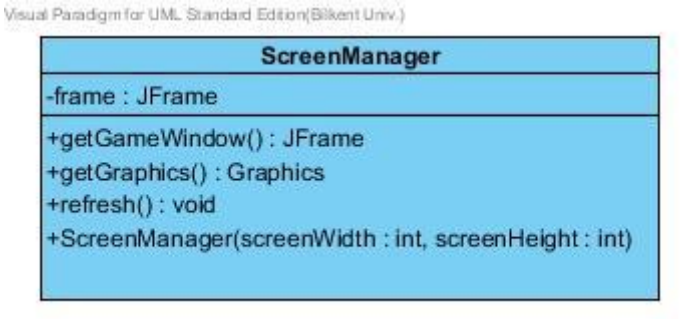
Methods:

- **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.

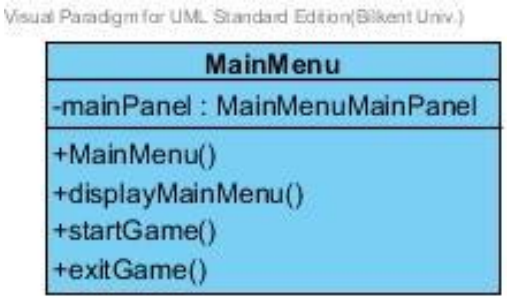
Constructors

- **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.

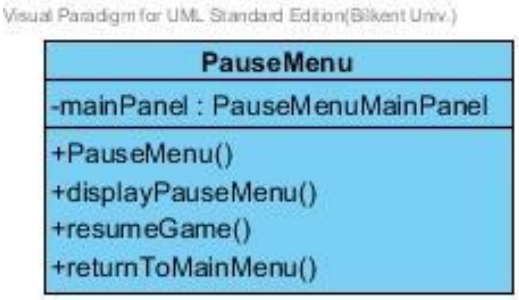
Methods:

- **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

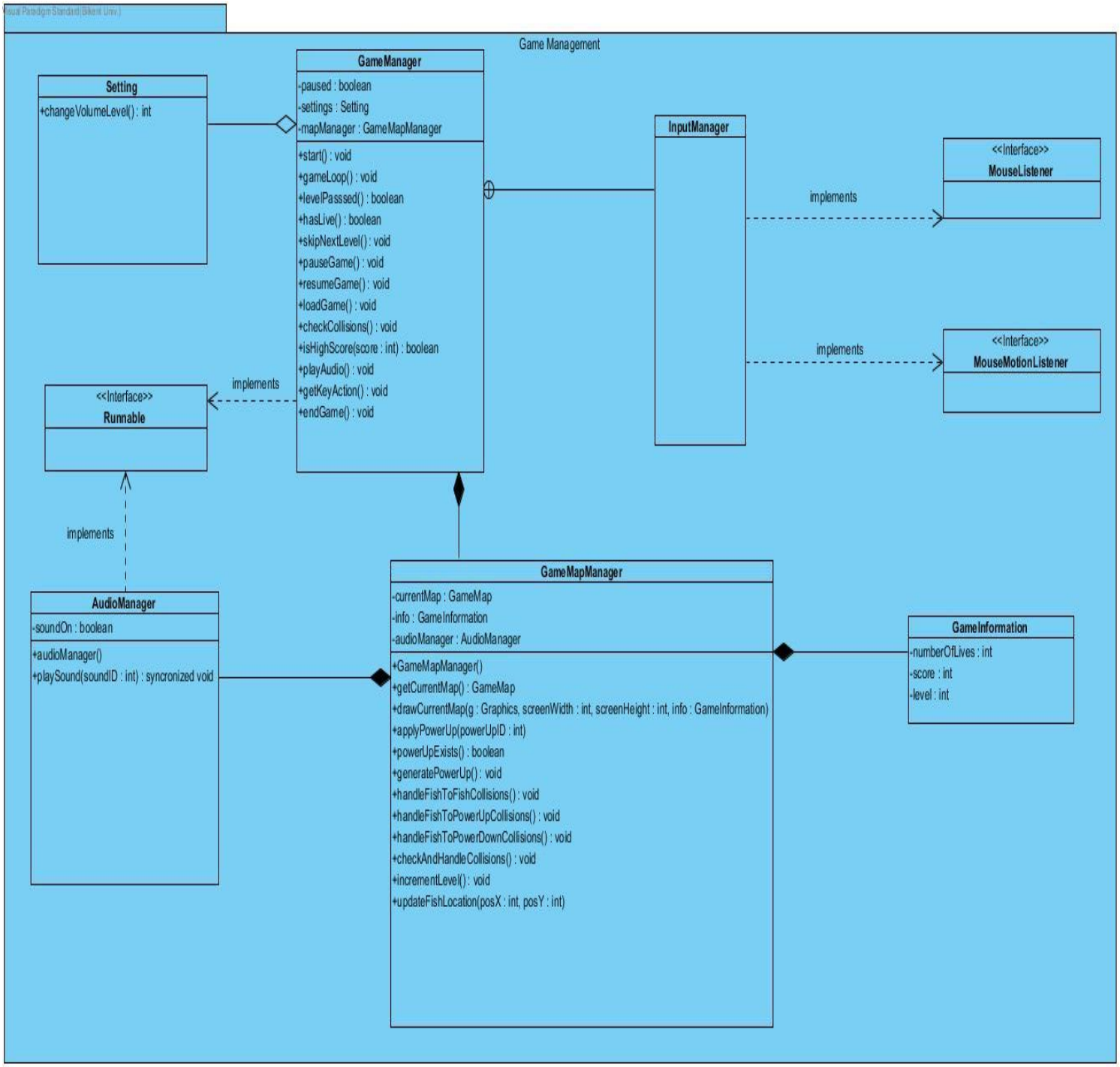
Methods:

- **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.

3.2. 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.



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.

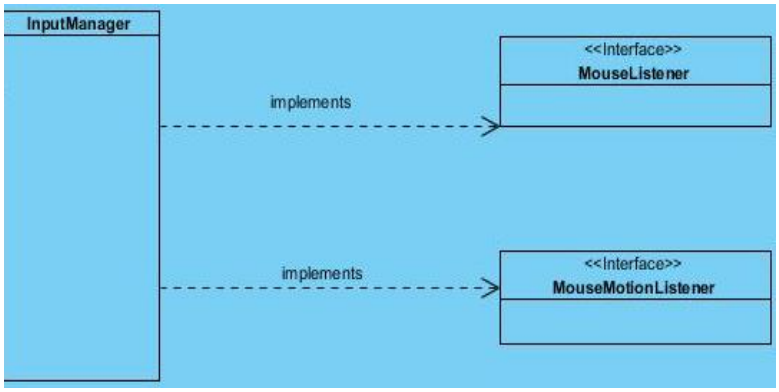
Constructors:

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

Methods:

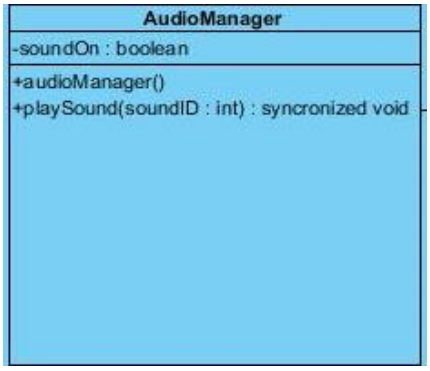
- **public void startGame():** It starts a new game through resetting game information stored on GameMapManager class.
- **public void gameLoop():** It runs a loop in which the system is updated continuously.
- **public boolean levelPassed():** It communicates with GameMapManager class to detect whether 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:

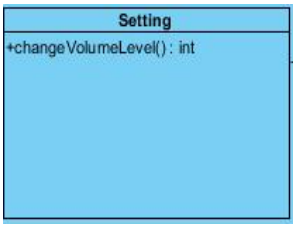
- **public AudioManager():** It initializes the soundOn object of this class which

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

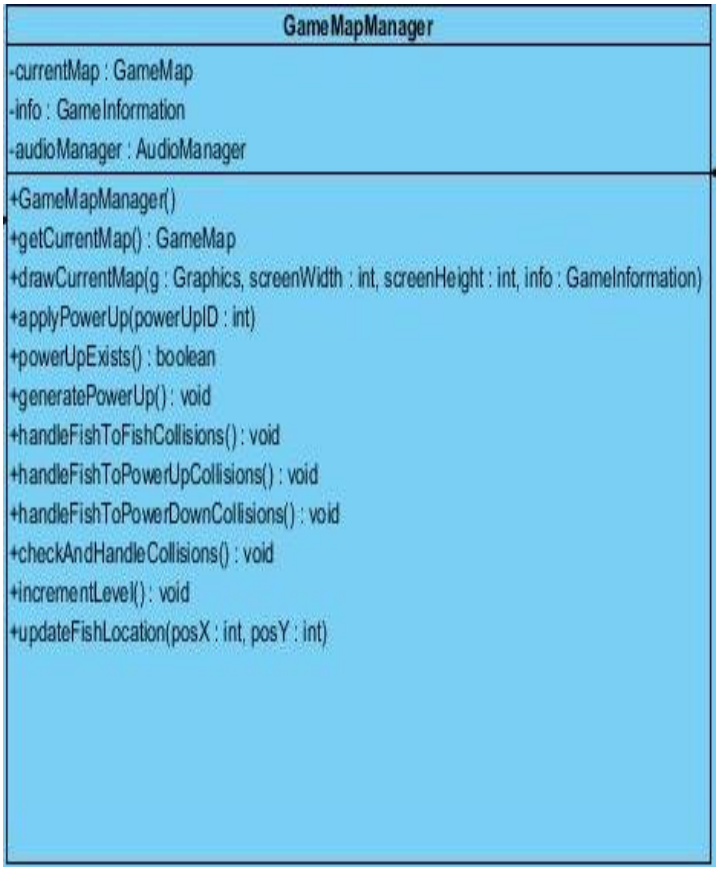


- This class is a simple property class that keeps the settings of the system. It is used by GameManager class.

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:

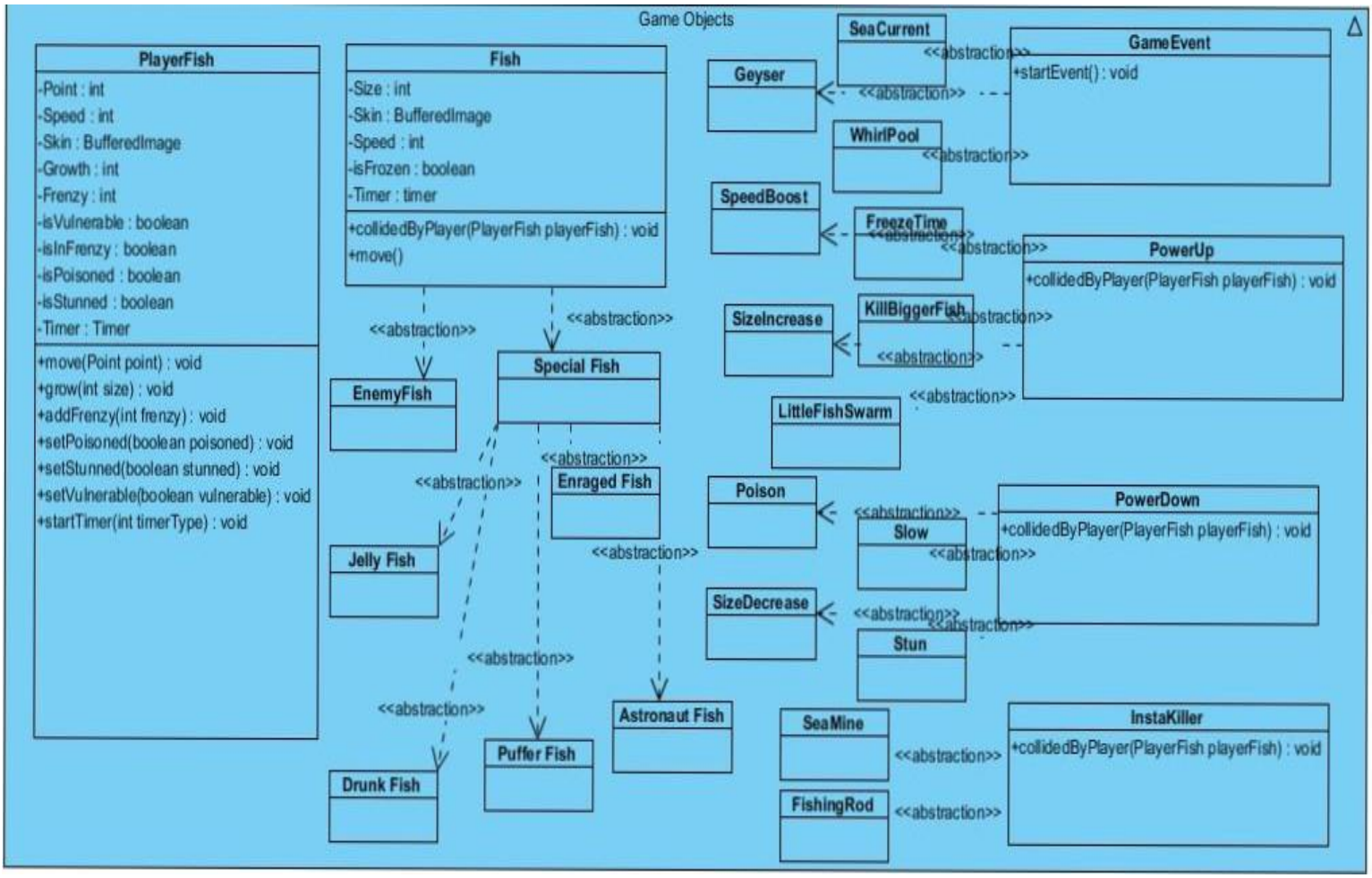
- **public GameMapManager():** It initializes a GameMapManager object with the default attribute values.

Methods:

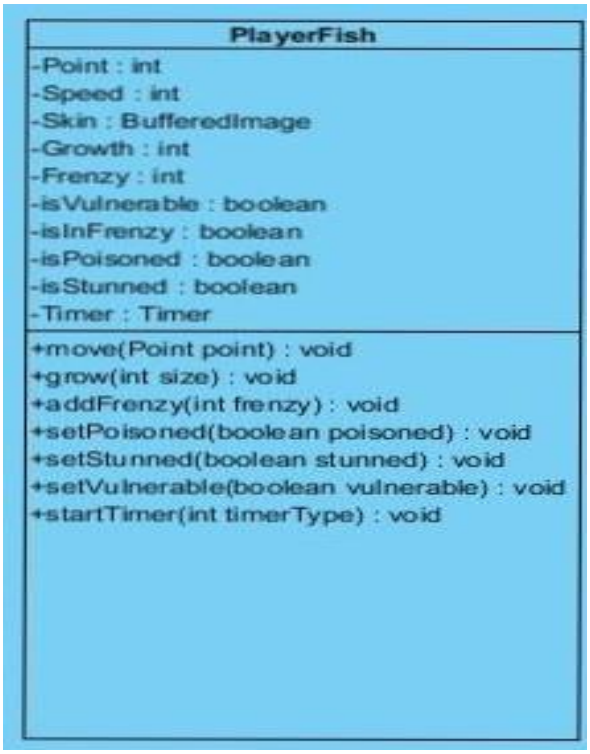
- **public GameMap getCurrentMap():** returns the current map which is processing by GameMapManager class.
- **public drawCurrentMap(Graphics g, int screenWidth, int screenHeigh, GameInformation info):** It draws the current map according to the given attributes.
- **public void applyPowerUp(int powerUpID):** 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.

3.4 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.



PlayerFish Class



Attributes:

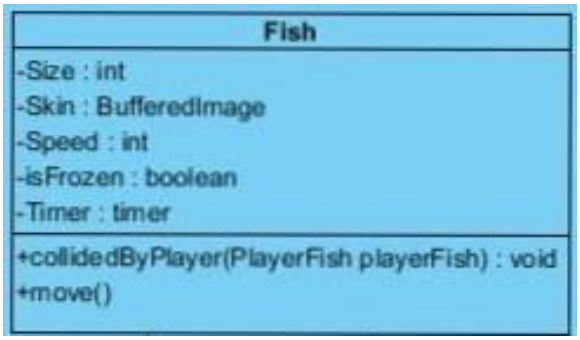
- **private int Point :** It keeps the record of the point which player's fish is at in each time. (like coordinate point)
- **private int Speed :** It is the record of player's speed that can change continuously.
- **private BufferedImage Skin :** It is a BufferedImage object which corresponds to a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

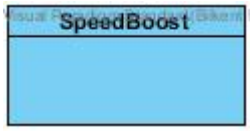
Methods:

- **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.

3.5. Detailed System Design

*Custom Fish: This class includes all the special fish explained in the previous section.