
# Object Oriented Software Engineering Project

## Design Report

CS 319 Project: Saving Humanity

Group 24

Can Bayraktar
Erdem Karaosmanoğlu
Mert Sebahattin Kutluca
Oğuzhan Karakahya

**Course Instructor: Uğur DOĞRUSÖZ**

**Table of Contents**

## 1.Introduction

### 1.1 Purpose Of The System

Saving Humanity is a game which we design to entertain people and cherish nostalgia to the ones that grew up with legendary atari game "Tank 1990". Our game is a remake of "Tank 1990" with some extra features. With that features our game promises more fun than its ancestor. Additionally with our multiplayer options, players can have fun with their friends in Saving Humanity.

### 1.2 Design Goals

**Usability:**

It is crucial for a user to feel comfortable in a game. Menu should help users to travel all features of game easily to attract the players. With "View Help" option you can see the button configurations of system. Help screen is shown while starting the game too. For Multiplayer Game options we will select the buttons so carefully to present comfortable setting for all of the four users in max.

**Functionality and Performance:**

Our main purpose for performance is a fluent gameplay. It is not hard to achieve because graphical requirements of game is low. Since it is a remake of old atari game, systems of today can easily run "Saving Humanity" at highest performance. Our purpose is to run the game with 120 FPS and smooth graphics. We will enable anti-aliasing for smooth graphics as we mentioned it in Analysis Report.

**Good Documentation:**

We aim to arrange our program well documented. We will try to provide JavaDoc documentation for all of the exposed classes and their exposed methods.

**Extensibility:**

Our game will be easy to extend because map files will be read from outside. So we can change map files easily. Even an editor program can be written to create new maps easily.

## 2. Software Architecture

### 2.1 Subsystem Decomposition

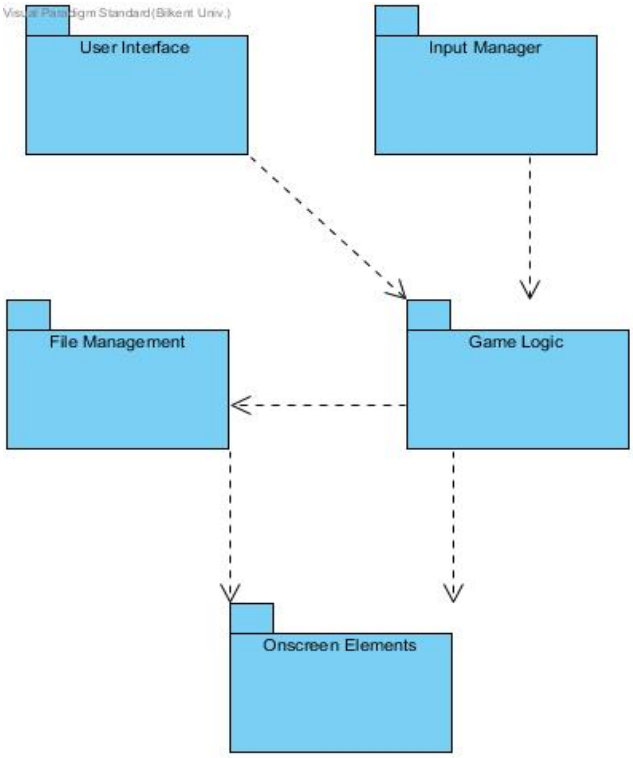



Figure 1

The architectural style of the system is a 3-Layer architectural style.It is also closed architecture(opaque layering).A layer's component can only communicate with the components in the same layer or the layer below.There are three hierarchically ordered layers. First one is the user interaction layer. It includes two components called “User Interface” and “Input Manager”.These components have classes which take inputs from user to initialize the game.
The second layer is the game logic layer. There are two components in this layer. “Game Logic” component handles all the game logic and mechanics with help from file management component. Finally, the third layer is called the “Display Layer” which represents all the entity objects of the game. All these object gets their information from the layer above.

### 2.2 Hardware/Software Mapping

Our project is going to be implemented in Java and because of that, the PC that is running the game is going to need the latest version of Java Runtime Environment. The game will require a keyboard to control the tanks and to enter a name whenever you beat a highscore in the game. Also to save the highscores we will a use .txt file which is supported in almost every operating system, therefore there isn’t any problem. The game won’t require a mouse and there won’t be any internet connection needed since our multiplayer is hot-seat.

### 2.3 Persistent Data Management

The disk is only used for storing game maps and keeping highscore file and the reads and writes are controlled by the program, meaning that there will be no concurrent writes. Therefore a simple file management system will satisfy our needs in term of data management instead of a database management system. Game maps will be stored as text files with predefined syntax and highscore file will contain name of players with their highscores. Map files will be read only while highscore file will be read and written consequtively by the program without any concurrency.

### 2.4 Access Control and Security

This game is played without any internet connection and there is only one type of user that is called as the player. As a result of that, we are not concerned with any security issues and also we do not need to implement and access control system.

### 2.5 Boundary Conditions

**Initialization**: Program needs to load proper images, sprites, map files and highscore file in order to run the game smoothly. When the user runs the executable file and start a game, all files will be read and game scene will be initialized.

**Termination**: Upon closing the game, each component related to the game will be terminated. Since there will be only one thread related to the program, it will be closed and program will exit.

**Errors**: Our aim is to provide best experince to users while playing the game, thus our main aim is to try recovering from errors as much as possible. However, if an error is crucial, such as failure of loading main game images, then game should be terminated. Thus, unless the error is not affecting the gameplay significantly, program will try to recover from erroneous situation.

## 3.0 Subsystem Services

### 3.1 Design Patterns

We have decided to use Singleton Pattern in our project since there are many classes that will only have one instance in our project. This patterns provides robustness to our project and makes sure that the system will operate more efficiently. The classes that use this pattern are GameEngine and InputManager, FileManager. These controller classes must be instantiated for once at most. We will implement a getInstance() method for these classes so that the one instance created can be accessed by the other classes. Additionally, to apply this pattern, we will implement a private constructor to these classes so that we can control the object creation. The one instance will be kept inside the class as a variable and all other classes that need to use the services of this class will need to get its one instance.

### 3.2 User Interface Subsystem Interface

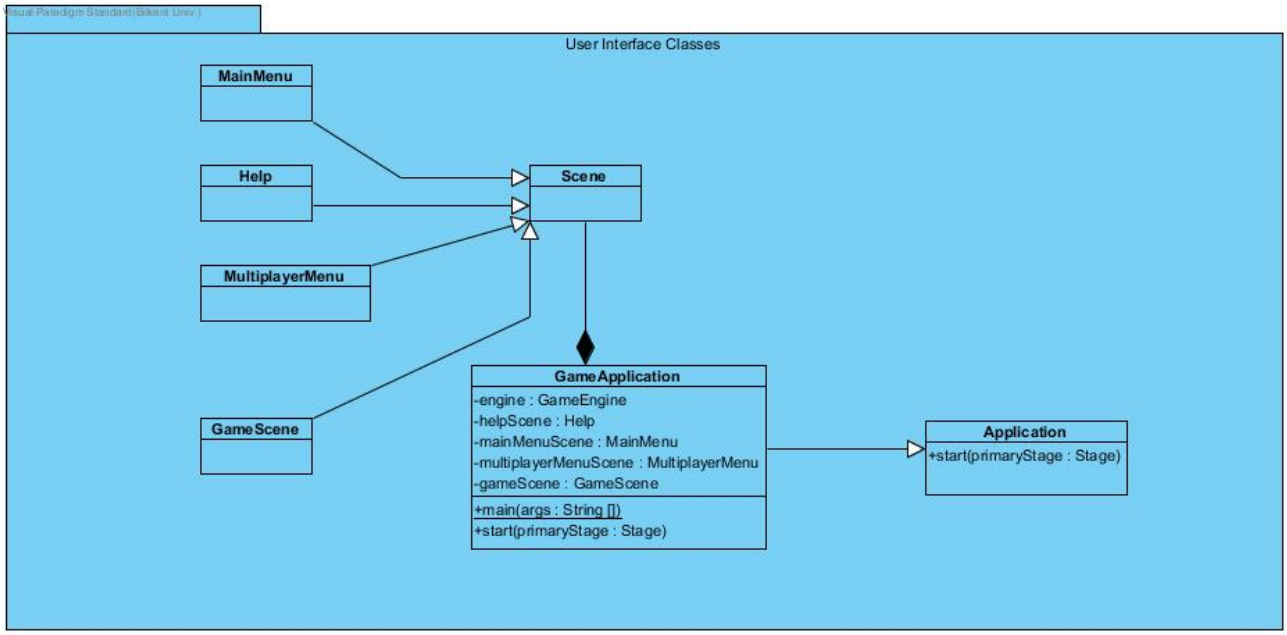



Figure 2

**MainMenu Class**

This class is responsible for demonstrating the main menu. It will contain the UI objects related to main menu such as buttons. It extends the Scene class of JavaFX.

**Help Class**

This class is meant to show Help menu from main menu or in game. This class is also an extension of the Scene class.

**MultiplayerMenu Class**

In this menu, users will be promted to choose a multiplayer menu with the number players that will play the game. Than the game will be started. This class is a child class of Scene class.

**GameScene Class**

This is the main Scene class that is responsible of showing the main game graphics. It will show the tiles, tanks, powerups and bullets with the informative sub panel on the right as drawn in the analysis report. GameScene class is a child class of the Scene class.

**Scene Class**

This is the corresponding class for Panel in Java swing library in JavaFX library. So they are used for all different type of menus. Since we have 4 special different menu, we will have 4 child class of this Scene class all of which are written specifically to represent these different menus.

**GameApplication Class**

This is the entry class of the application. It contains the main method, and it is the main UI class that contains all different types of Scene classes. It overrides the start() method of the

Application method that is another built-in class of JavaFX library.

**Attributes**

- **engine : GameEngine:** It contains the game engine instance, the game initializations, collision checks and graphical updates will be invoked by the game engine.
- **helpScene : Help:** It contains the HelpScene child class of Scene. So when clicked, this help page will be displayed on the screen.
- **mainMenuScene : MainMenu:** Contains a pointer to the MainMenu object to display the main menu of the game. It is the first menu upon initializing the game.
- **multiplayerMenuScene : MultiplayerMenu:** Contains the scene for multiplayer menu
- **gameScene : GameScene:** Contains the main scenery of the game. Each entity object will be drawn on this scene during the game.

**Methods**

- **start(primaryStage : Stage) :** This method is defined in the Application class. It is the main method for instantiating the stage. It will be overridden to display the initial scene and later on, upon use inputs it will switch between different scenes.

**Application Class**

This the built-in class of JavaFX library. Basically it is used to enter the program and instantiate the GUI objects inside the program.

## 4.0 Low-level Design

### 4.1 Object Design Trade-Offs

We have decided that some of the design goals are more important than others therefore we had to have some trade-offs. The first trade-off is between run-time efficiency and ease of writing. Instead of checking collisions between tanks and tanks, tanks and bullets, tanks and tiles, bullets and tiles, etc. we have decided to write a single collision checker method and pass two game objects to it. This would check collisions between two tiles as well which is pointless. Since computers are fast nowadays and we don’t have too many game objects we thought that this would be a good idea.

Another trade-off is between using memory efficiently and flexibility. We have decided to use static variables where many classes needed the same variable and would constantly using it. We have chosen to use memory more efficiently by using static varibles but they tighten the dependencies between classes and object therefore our project might not be very flexible in terms of using this issue.

## 4.2 Final Object Design

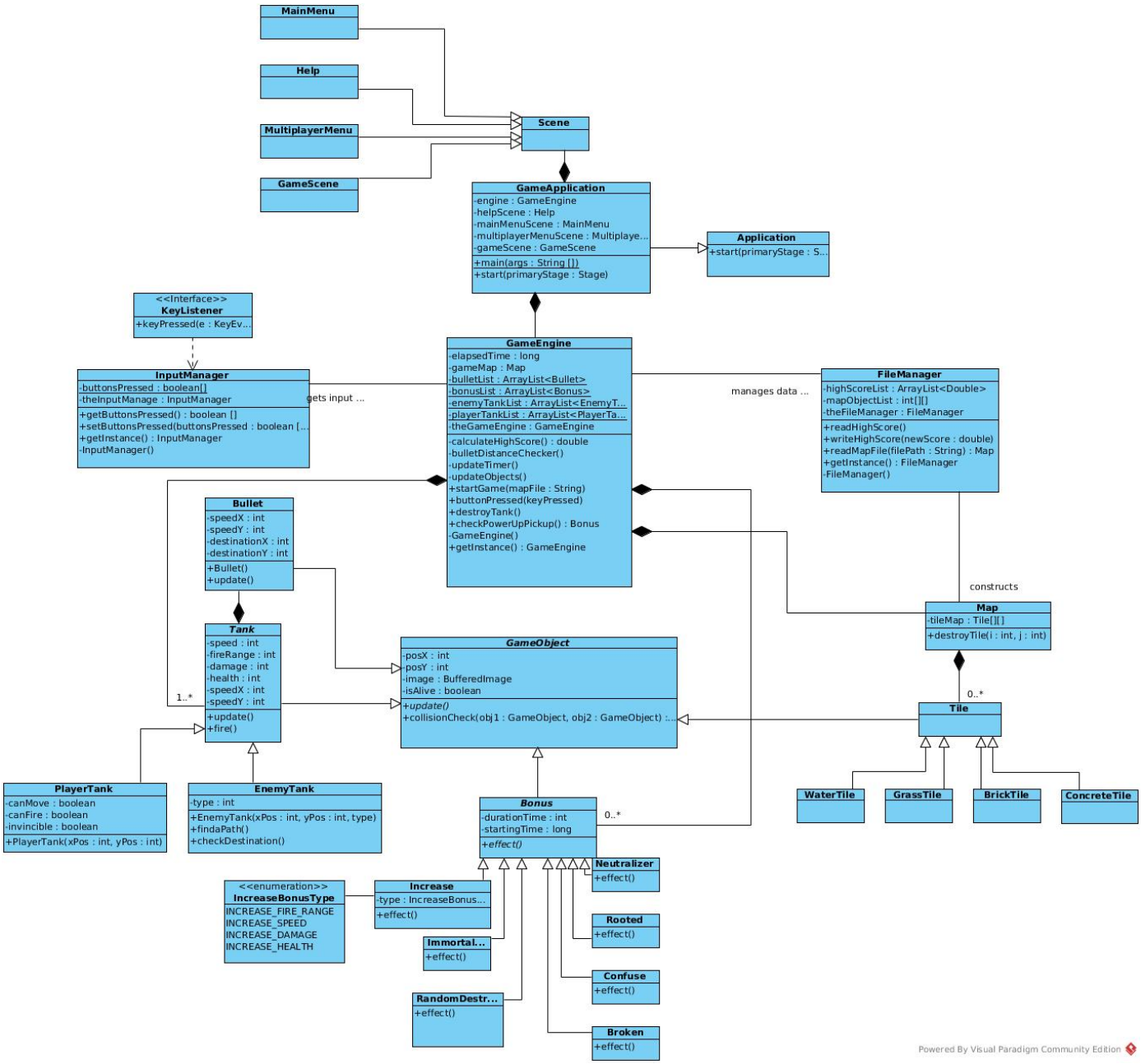
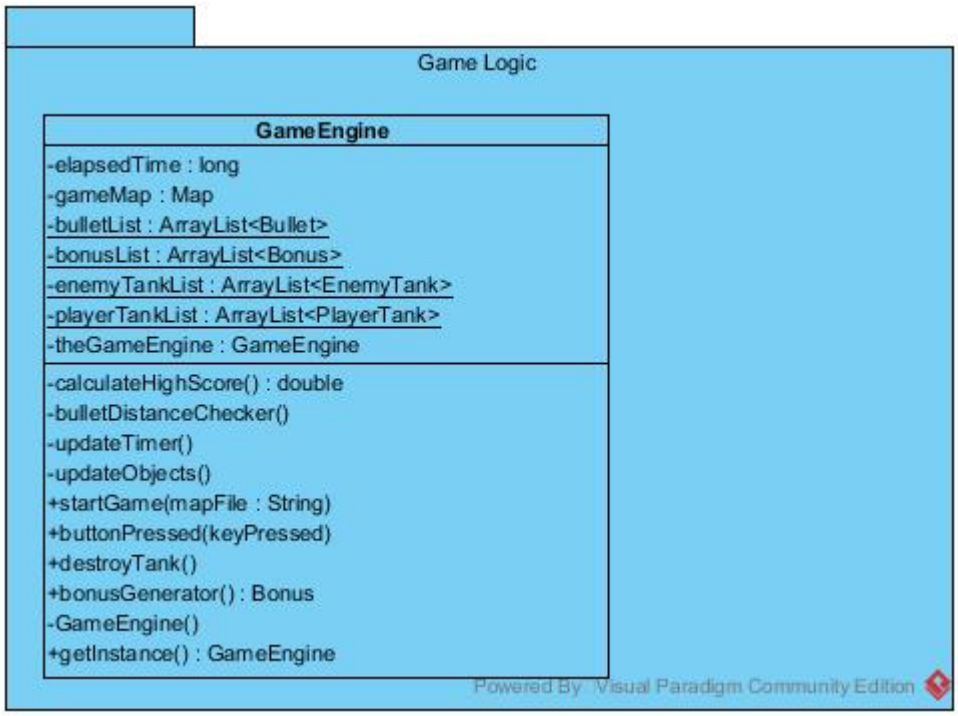



Figure 2

## 4.3 Packages/Subsystems

### Game Logic Subsystem

The Game Logic Subsystem manages the all other subsystem. For example its takes the information of the a level from File Management Subsystem, takes the information of game objects from Onscreen Element Subsystem then it gives these User Interface Subsystem to draw them.




**Game Engine Class**

**Attiributes**

- **elapsedTime : long:** It starts with zero when the game is started. Its type is "long" because its unit is milliseconds and it might br so huge.
- **gameMap : Map:** This object is a maze and it consist of a two dimensional array. There is only one Map object at an particular time. It is constructed when a new level has started and it is modified accordingly "File Management Subsystem" when the level has been completed and new one is constructed.
- **bulletList : ArrayList<Bullet> :** It contains the bullet object which are on the screen and active. The size of this list can not be more than tankList's because tanks can not fire when their bullet has been fired, has been terminated.
- **bonusList : ArrayList<Bonus> :** It contains the bonus objects on the screen.
- **enemyTankList : ArrayList<EnemyTank> :** This list contains all enemy the tank in current level. When a tank destroyed it will also removed from this list. It will be empty in multiplayer mode.
- **playerTankList : ArrayList<PlayerTank> :** This list contains the player tank in current level. Its size will be one in single player mode and in multiplayer mode the size can be two, three or four; it is determined by "Input Manager" subsystem.

**Methods**

- **calculateHighScore() :** When the all levels are finished, this method will check the

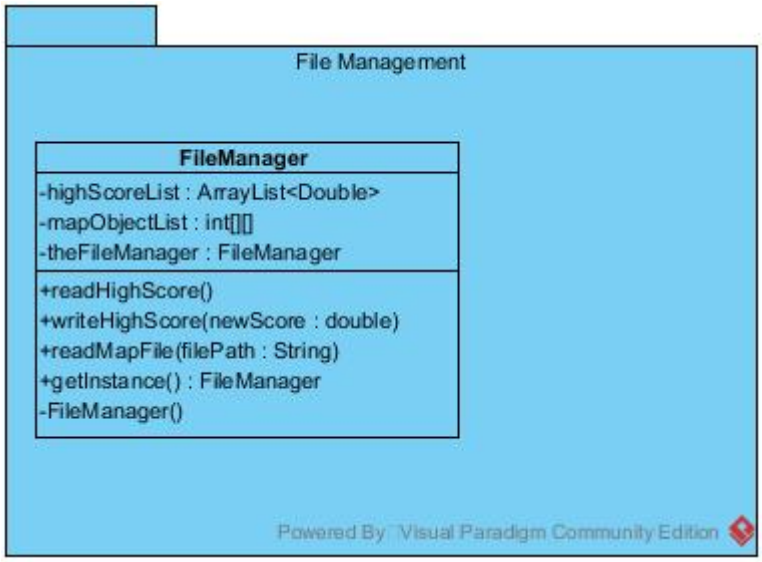
remaining health of player tank and elapsed time, then it calculates a score. Then it looks the highScoreList in File Management Subsystem and if the score is in top ten this method calls writeHighScore(double) method of File Manager class.

- **bulletDistanceChecker(Bullet) :** It checks whether the bullet reaches its destination, if it is, its terminates the bullet.
- **updateTimer() :** It increments the elapsedTime.
- **updateObjects() :** It calls the update methods of game objects.
- **startGame(mapFile : String) :** It creates tank objects and fills, enemyTankList and playerTankList. It also creates the gameMap object accordingly File Management Subsystem.
- **buttonPressed(keyPressed) :** It takes the keyboard input from InputManager Subsystem to determine what playerTank do.
- **destroyTank() :** It removes tank objects from enemyTankList or playerTankList if their health is lower than or equal to zero.

**bonusGenerator() :** When an enemy tank destroyed, this method is called. It generates a random bonus on the position where the tank has been destroyed and it adds this bonus into the bonusList. If the bonus is taken or its lifespan is over, this method remove the bonus from the bonusList.

**File Management Subsystem**

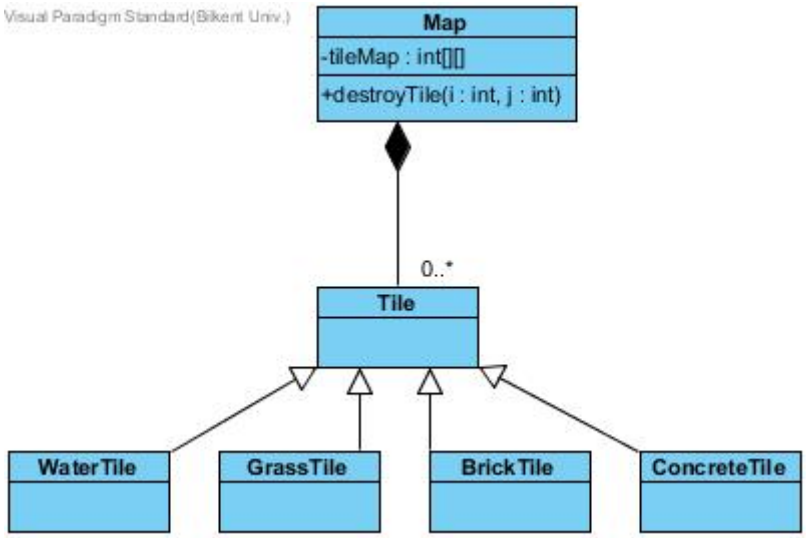
This subsystem is responsible for all storing and loading operations. It loads the map File and high scores from hard disk and it also changes the high score list.



**File Manager Class**

This class is the mere class of File Management Subsystem. Its readHighScore() methods read the file and fill the highScoreList. It can also add a new high score with writeHighScore(newScore : double) method but this method does not change the highScoreList, in order to change this list readHighScore() method should be recalled.

Specifications of maps are stored in hard disk. readMapFile(filePath : String) method fill the mapObjectList array by reading the file. Game Logic Subsystem takes this array to construct "gameMap" object.




**Map Class**

This class includes two dimensional Tile array which creates the maze where tanks cruises. If a tile is breakable, when it has broken, it is deleted from the array and it becomes null, destroyTile(int, int) method performs this.

**Tile Class**

There are four different types of tile and this class is the parent class of them.

**WaterTile Class**

Tanks can fire over this tile but they can not go over.

**GrassTile Class**

Tanks can fire over this tile and they can go over. However when a tank is on this tile, it camouflage this tank.

**ConcreteTile Class**

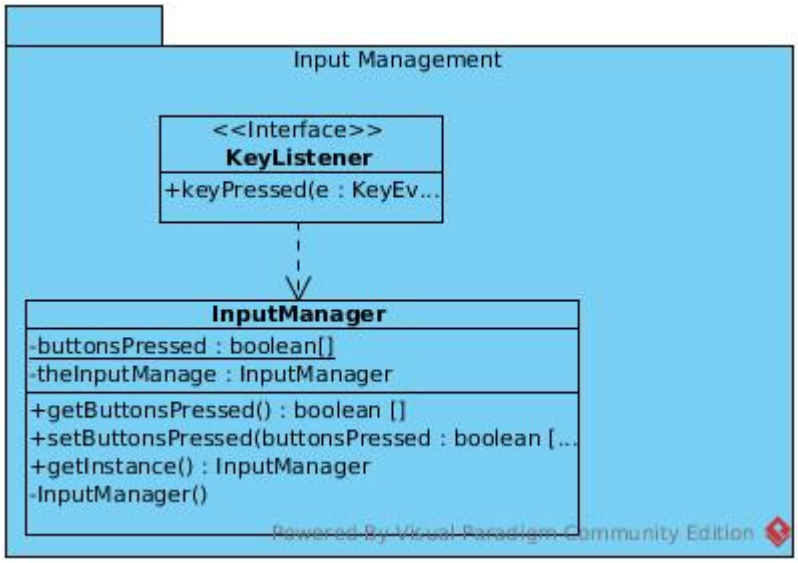
Tanks can not traverse over it and it can not be broken by fire.

**BrickTile Class**

It is breakable but tanks can not traverse over it.

**Input Management Subsystem**

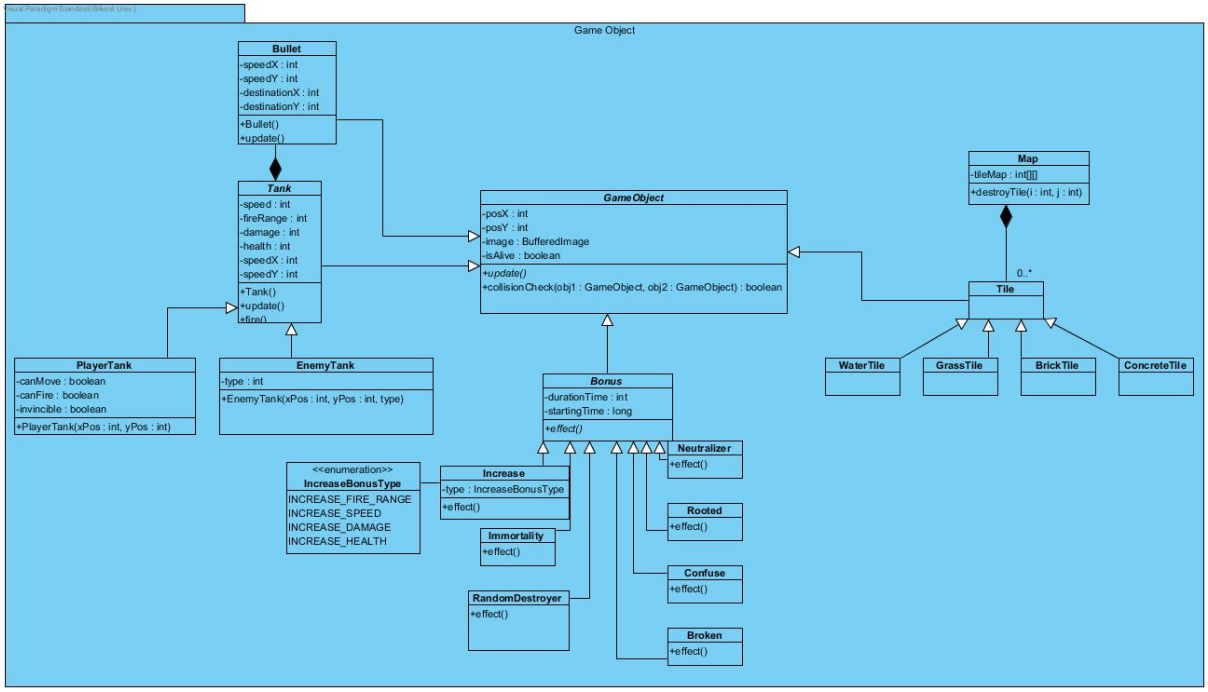
This subsystem provides connection between program and user. Takes user's commands and manages other subsystems according to these commands.



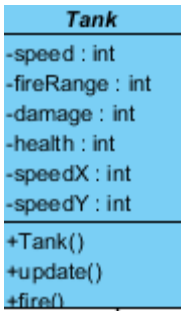

**Input Manager Class**

This class controls which buttons are pressed by user. Change default false booleans into true and sends this information to the game engine. This class interfaces KeyListener class from Java. All of the buttons, useful for game, are boolean variables and stored in array list which called as buttonListener.

**GameObject Subsystem**



**Tank Class**




Attributes

-speed : int: It shows how many units tank should move per move command.

-fireRange : int: Defines the range of tank's bullet. Bullet can go this amount of pixels at maximum

-damage : int: Explains how many health will go away when tank's bullet hits other tank.

-health : int: Shows tank's remaining health. It can change with get shot from enemy tank and getting a health increasing bonus.

-speedX : int: Shows the speed of tank on X coordinate.

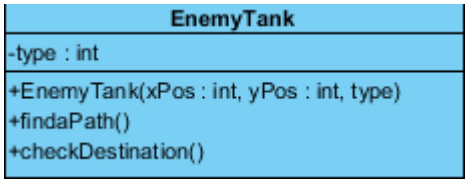
-speedY : int: Shows the speed of tank on Y coordinate.

Methods

-update(): We use this method when we want to change tank's position.

-fire():With this method tank fires a bullet. So, this method creates an new Bullet object.

**Enemy Tank Class**




These type of tanks occur only at singleplayer mode. Their properties change according to level. It is child class of Tank class so it has Tank's attributes and methods. Additional properties are:

Attributes:

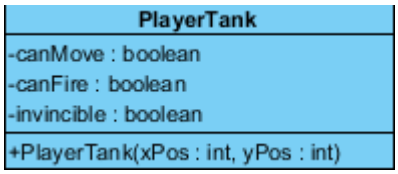
type : int: It used ID of enemy tank. Helps us to select to apply bonus on which enemy tank.

Methods:

findaPath(): This method draws a path for enemy tanks. If there is a brick tile or water tile on the way, methos changes tank's direction.

checkDestination(): Checks whether player tank is on range or not. If it is in range fires a bullet to the player tank.

**Player Tank Class**




Tank which player use on Singleplayer mode and every tanks in the Multiplayer mode. Moves according to player's commands. It is child class of Tank class and additional attributes can be seen below:

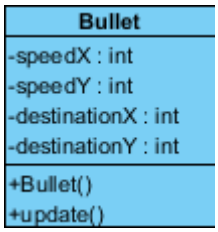
Attributes:

-canMove(): boolean: Controls whether tank can move or not. Default value is true and it can be changed to false with Rooted bonus.

-canFire(): boolean: Controls whether tank can fire or not. Default value is true and it can be changed to false with Neutralizer bonus.

-invincible(): boolean: Controls tank can be seen by enemy tanks or not. Default value is false and it can be changed to true when tank enters grass tile.

**Bullet Class**




Bullets are created by fire() method of Tank class.

Attributes

-speedX : int: Shows the speed of bullet on X coordinate.

-speedY : int: Shows the speed of bullet on Y coordinate.

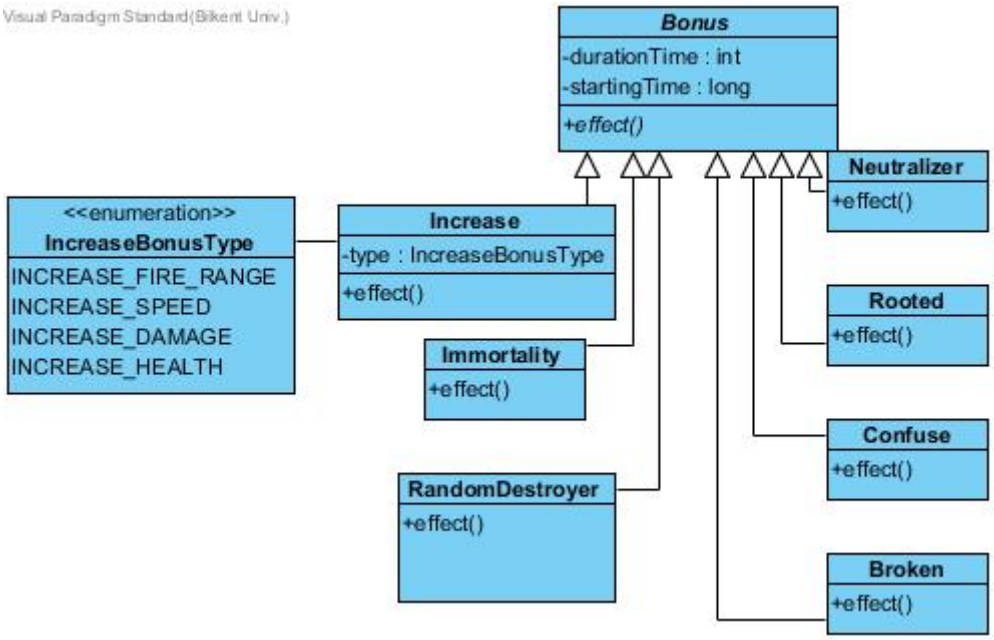
-destinationX : int: Shows the location of bullet on X coordinate.

-destinationY : int: Shows the location of bullet on Y coordinate.

Methods

-update(): Updates bullet's location continuously until a collision is detected or reaching range of bullet.

**Bonus Class**




The bonus class is an abstract class that has two attributes which are durationTime and startingTime. durationTime is an integer that stores how many seconds a bonus is going to last. startingTime is a long that will hold the system clock value when the bonus is collected by a tank which will be crucial when calculating the bonus' duration time. It also has an abstract method called effect() to simply use polymorphism since each children of Bonus has a method called effect() in them.

The Bonus class has seven child classes each representing a different type of bonus except the Increase class. As stated above each class has its own effect() method that are defining the effects they do on tanks whenever they have picked up. Different than other children the Increase class has an attribute called type which takes the enumator type IncreaseBonusType. We have decided to use enum here as the game will have four different increases that are fire range, speed, damage and health.