CS 319 - Object-Oriented Software Engineering

System Design Report

Panzer17

<u>Group 2-C</u>

Pınar BAYATA

Burak SİBİRLİOĞLU

Ndriçim RRAPİ

Contents

1. Introduction

1.1 Purpose Of The System

Panzer17 is a re-constructed version of the old classic Tank 1990 game. While protecting the essence of the original game, what we try to achieve is to give players an alternative taste of what could happen with additional features. Those features include different types of enemies and a rival castle we can attack as the player to win the game without destroying all of the enemy tanks. The user needs to adapt himself/herself to new conditions that comes with each map, which all have exclusive features.

1.2 Design Goals

It is a better approach to think before act, and to apply this policy we ensured to make the system design before implementation. In our design we divided our goals to four:

1.2.1 End User Criteria

Panzer17 must provide a familiar look for those who played the original Tank90 game, which was a basic game therefore graphical user interface should be easy to understand. Player should use the controls and menus of the game easily and also player should understand the map as fast as possible to start the game.

1.2.2 Extensibility Criteria

We will try to have many simple classes to have a chance to improve the game after the demo according to tests and user ideas. Having such kind of implementation will allow us to change attributes of main items without changing the main classes. This will make the game extendable and reusable. Also according to our opinion maintenance and solving some runtime errors will be easier.

1.2.3. Performance Criteria

The game should work fast enough not to create discomfort. As a classic console game, requirements of Tank90 was not high. We aim to keep the requirements in Panzer2017 as low as possible to have an efficient performance on every system that the game is played.

1.2.4 Trade Offs

Game have sound and some graphic features like moves of tanks, bullets and destroy of tanks or walls. These features should not decrease the speed of gameplay, to guarantee the enjoyment of the game.

1.3 Overview

Panzer17 starts with the top view of a map and tanks. The map consists of various bricks such as one brick can be collapsed with one shoot while the other one should be shot three times. There are two castles, one for the enemies and one for the player. The castles are surrounded by the easily destroyable bricks. There will be 3 levels and in each level, map's design is changed so that it will be hard for the player to move and catch an enemy. Enemies are classified in themselves. Like the bricks, some of them have to be shot more than the other ones. Heart counter starts to count from 3 and decreases the value whenever player is shot by an enemy. Apart from that, point counter will start from 0 and adds the points as the player shoots enemies. An enemy will not be able to shoot other enemy. An enemy's tank will be different from the player's. Enemy's position will be generated randomly in every level. The goal is to play all the levels and reach the castle in every levels.

2. Software Architecture

2.1. Overview

Maintaining and implementing a project needs manageable small subsystems. In this section we will divide project implementation into 3 subsystems which are game management, user interface and game entities. Main reason of this decomposition is to reducing the complexity of coding however the small subsystems will increase the complexity of interaction between subsystems.

2.2. Subsystem Decomposition

Creating small subsystems will increase our ability to modify, control and implement the project. Dividing the project into small parts will also increase the effectiveness of coding and implementation therefore performance of the project will be better. To meet this functional and non-functional requirements of the application, decomposition idea will be used.

Different aspects of the project will be controlled by different subsystems and the subsystems will be connected by using gameManagement and Panzer17 classes. General organization of UserInterface, GameManagement and GameEntities subsystems can be seen in Figure-1. Classes that are working together also need to communicate with each other to being synchronized. Therefore connection of classes and different subsystems can be seen in the Figure.

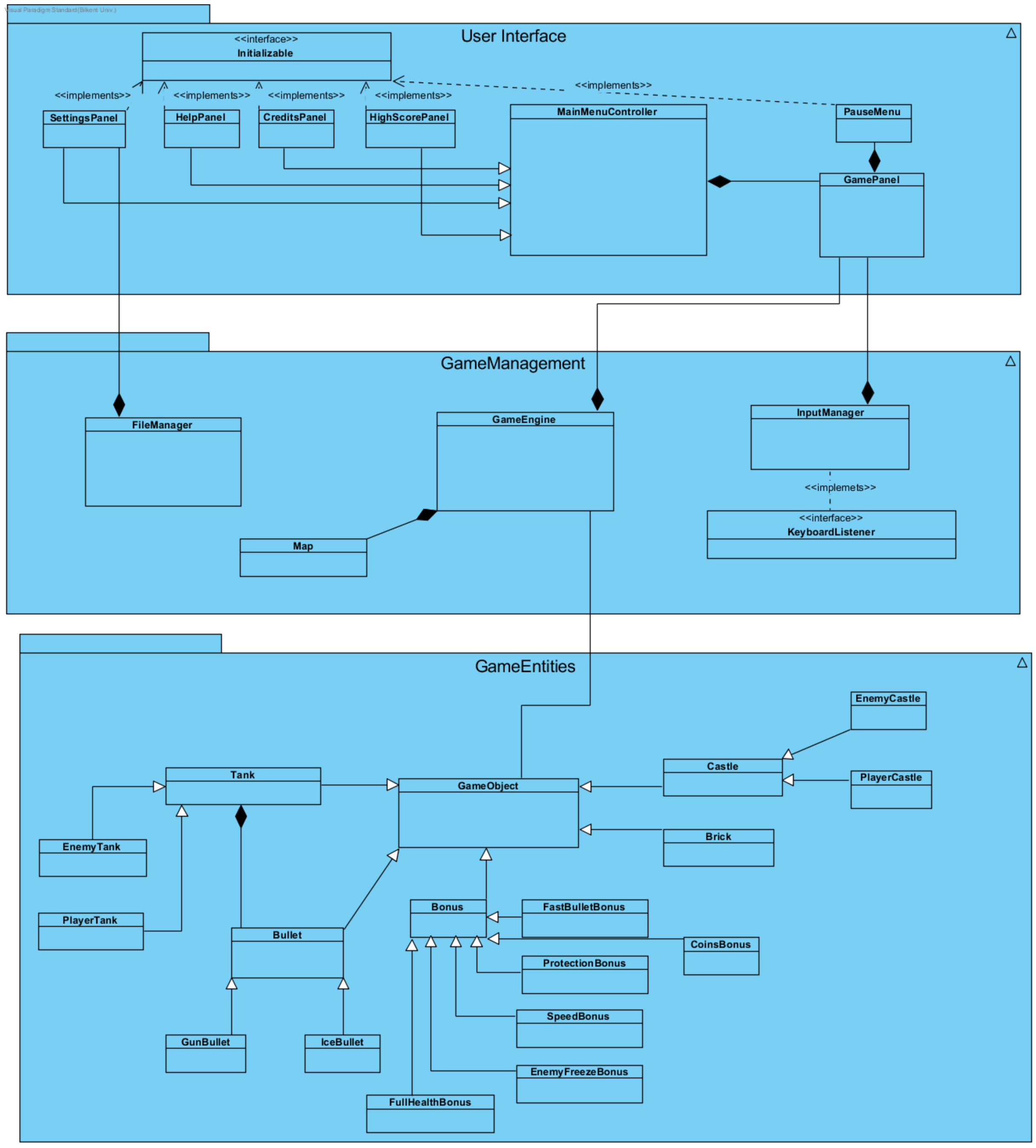


Figure 1: Subsystem Decomposition

2.3. Architectural Styles

2.3.1 Layers & Model View Controller

Model View Controller mechanism suggest 3 different implementation layers. We defined our subsystems according to this idea and therefore we have three different implementation layers. User Interface, Game Management and Game Entities layers will be used in the project and architectural style will follow this hierarchy. As the User Interface controls the input and output mechanism of the project, this layer will be the top layer. User Interface layer can use Game Management layer which is the controller of the MVC idea and this layer will connect game objects and User Interface of the system. At the lowest level Game Entities layer will be implemented and this layer will be the basis of the Project by containing all required objects and mechanism. Game Entities layer will be the Model of the project. This kind of architectural design will allow us to change different aspects of project without affecting other layers and subsystems. In this way, this principle will increase the effectiveness and management of implementation process of the project. It will also increase the maintainability and flexibility of the system.

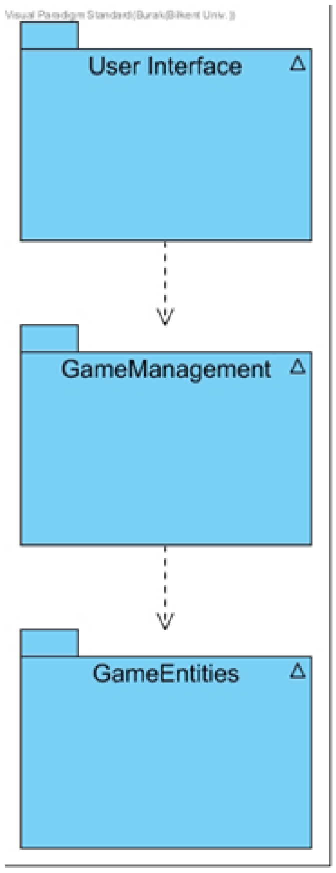


Figure 2: Decomposed System Layers

2.4. Hardware / Software Requirements

Panzer17 project will be implemented by latest edition of Java programming language. By using the portability of Java, the game will be independent from operating systems only Java Runtime Environment will be enough. In other Word our game will be designed to work in all kinds of systems. However, the system has no database to store the data, because the data will be constant, game data will be stored in a simple text format instead of database. Therefore environment that game will be operated should support at least one kind of simple text editor. Apart from that java complier need to be used to compile the game in the environment.

On the other hand, Panzer17 requires basic hardware components. Any standard computer with a monitor and basic keyboard will be enough to run the game. Because the game will be offline and single player there is no need for internet connection.

2.5. Persistent Data Management

Since the game data will be constant and will not be changed during the gameplay, we decided to not implementing a database like structure. İnstead of database, map data of levels, high scores of player and setting data will be stored in simple text file format. Apart from that this data will be crucial to starting the game, therefore we will check the text files at the starting stage of the runtime and according to result systems will respond. If any error will be detected in the data files, system will stop execution and inform the user about the error.

On the other hand, image and sound data will be stored as simple formats to increase the performance of the game by decreasing the loading time of the game.

2.6. Access Control and Security

Panzer17 game doesn’t require any internet connection therefore there is no need for complex security conditions. However the game will need a user name in the beginning of the game to hold the high score data. Therefore we need to check the previous user names to prevent coincidences and guarantee that the score record is safe. Apart from that we won’t allow user to change the map or basic game objects, we restrict the access of game objects by using the settings panel and user can change only the settings that are allowed in the panel.

2.7. Boundary Conditions

- ***Initialization***

The game will have no installation process because the game will be published as executable jar file. This will increase the portability and ease of use.

- ***Termination***

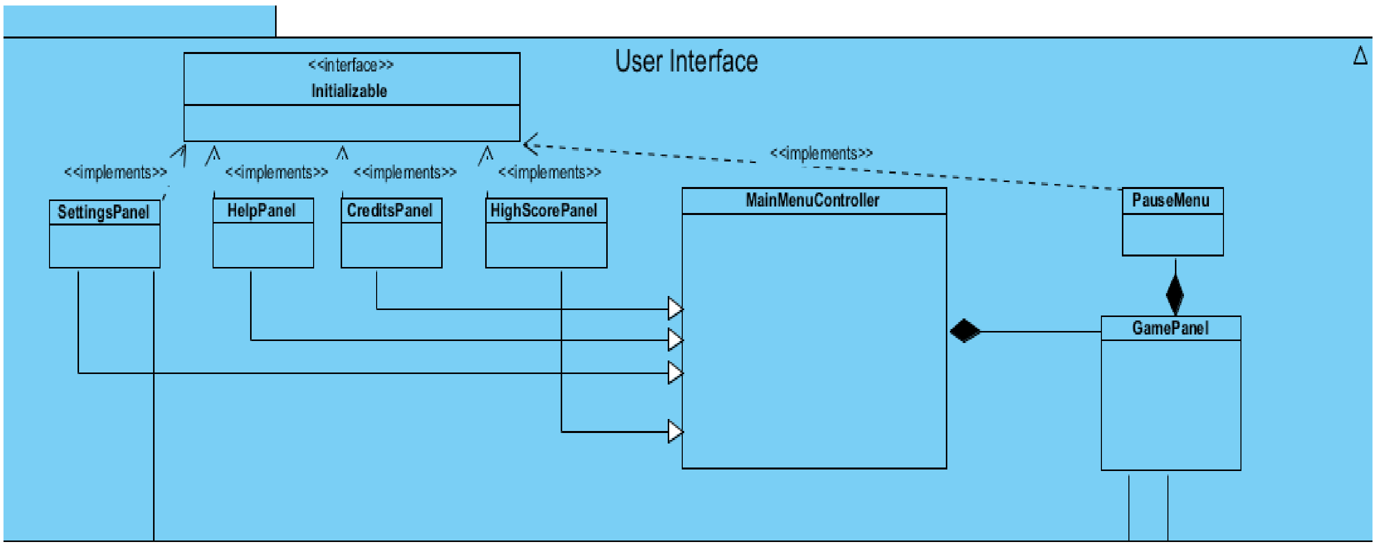
There will be three case to terminate the game. First case is the crucial error case which is the non-availability of data files. In this case the system will create an error message and terminate the runtime immediately to secure the other data and the system. Other two cases will be user controlled. User can choose to close the game from the main menu or the pause menu during the game play. Both of these menus will have quit button which terminates the game after asking about user approval.

- ***Error***

Error cases will be differently controlled, if the sound effects or the high score data have an error on loading or during gameplay, game will continue to work without these components because they won’t affect the functionality of the game. However an error in the main data files, such as map data, or main code blocks like gameManager class will create a runtime error message and terminate the program immediately.

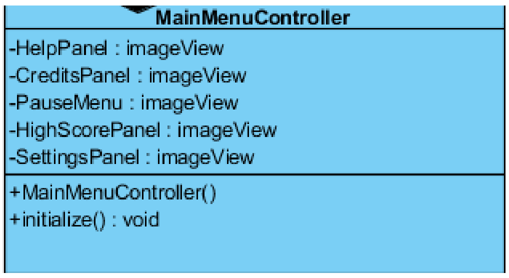
3. Subsystem Services

3.1 User Interface Subsystem Services



The user interface decides what to show on the screen when the application is running. It starts with the main menu, provides several different menus and the gameplay itself. The user encounters with a Main Menu when he/she opens the game. There are four buttons which are used to open new screens such as the player can view Credits, High Scores, Settings and Help. On the other hand, there is a Pause Menu which is to pause and play or exit. All changes will be updated in the Game Management level. So this subsystem is also interacts with the Game Management Subsystem. In this section, we defined the User Interface Subsystem classes.

3.1.1 Main Menu Controller Class



Main Menu Controller is the welcoming scene. When a player opens the game, this is the first panel to be shown in the screen. As it is said in the above, user can view Credits, High Scores, Help and Settings apart from playing the game. With the help of this class, player can switch to the other screens and go back to the Main Menu when he/she is done with the screens.

This class makes use of JavaFX's Initializable[1] class which accesses an FXML file containing all of the layout information. This will allow to add more design specifications in the way that this Menu looks like.

Also in this way we can improve on design details by making use of the FXML-s.[1]

Constructor:

public MainMenuController (): Initates a panel which is going to hold all of the menu buttons neccassary to implement main menu functionality.

Attributes:

private imageView helpPanel: with the help of this attribute, we will have an image of help panel in the main menu.

private imageView creditsPanel: with the help of this attribute, we will have an image of help panel in the main menu.

private imageView pauseMenu: with the help of this attribute, we will have an image of pause menu in the main menu.

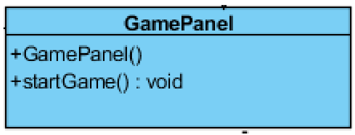
private imageView highScorePanel: with the help of this attribute, we will have an image of pause menu in the main menu.

private imageView settingsPanel: with the help of this attribute, we will have an image of pause menu in the main menu.

Methods:

public void initialize(): by using JavaFx’s facility this method initializes and sets all the positions, images of the all panels.

3.1.2 GamePanel Class



When the user clicks on the **playButton** in **MainMenu**, **GamePanel** will be shown. This class will design the main game screen view. This view will consist in a panel with the game's map in the center and a status bar at the bottom of the screen which will show health bars collected points and bonuses as the user keeps on playing the game.

This class makes use of JavaFX's Initializable[1] class which accesses an FXML file containing all of the layout information. This will allow to add more design specifications in the way that this Menu looks like.

Also in this way we can improve on design details by making use of the FXML-s.[1]

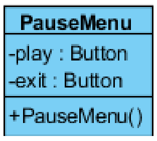
Constructor:

public GamePanel (): The constructor for GamePanel. This constructor creates a panel which is going to hold **timeline, timer,** and **another** attributes.

Methods:

public void startGame(): This method is invoked from the GameEngine class and this method after the initialization of the objects, shows these objects on the screen.

3.1.3 PauseMenu



The game can be paused during gameplay. When the player tries to pause the game, all tanks and bullets stay in the same place and game progress won't be lost. Also when player start again the game will continue immediately. Apart from that player can reach help panel from pause menu. Also player can quit the game by this menu.

This class makes use of JavaFX's Initializable[1] class which accesses an FXML file containing all of the layout information. This will allow to add more design specifications in the way that this Menu looks like.

Also in this way we can improve on design details by making use of the FXML-s.[1]

Constructor:

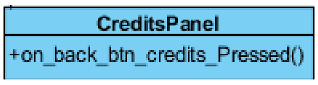
public PauseMenu (): The constructor for Pause Menu. This constructor creates a panel which is going to hold **playButton**, **backGround** and **exitButton**.

Attributes:

private Button play: The button to return to gameplay.

private Button exit: Provides user to close the game and return to desktop.

3.1.4 CreditsPanel Class



After the game is completed, user can see the credits panel and this panel have the information about, developers which can be used to communicate with, designers to give some new ideas which can develop the game.

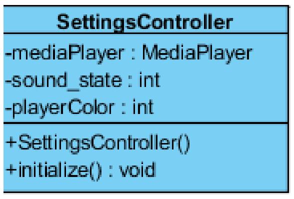
This class makes use of JavaFX's Initializable[1] class which accesses an FXML file containing all of the layout information. This will allow to add more design specifications in the way that this Menu looks like.

Also in this way we can improve on design details by making use of the FXML-s.[1]

Methods:

public void on_back_btn_credits_pressed(ActionEvent e): This method describes the action which will be performed after the back button is pressed.

3.1.5 SettingsController Class



The options menu that allows the user to change certain settings. Apart from the default settings, player can change the settings of the game. As a basic console game, the game hasn’t got many settings. Player can only enable or disable the sound of the game and also can choose the color of the tank.

If the player wants to turn to original settings of the game, player can choose the default settings.

This class makes use of JavaFX's Initializable[1] class which accesses an FXML file containing all of the layout information. This will allow to add more design specifications in the way that this Menu looks like.

Also in this way we can improve on design details by making use of the FXML-s.[1]

Constructor:

public SettingsPanel (): The constructor for Settings Panel. This constructor creates a panel which is going to hold all of the other components such as **backButton** and **resetBtn** attributes.

Attributes:

private MediaPlayer mediaPlayer: this attribute is needed when the player presses the sound button. With this initialization, we can play with the sound if it is on or off.

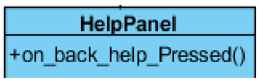
private int sound_state: by looking at the value of the integer, one can say the sound is on if the value is 1, and the sound is off when this integers’ value is 0.

private int playerColor: this attribute identifies the color of the tank by its value such as ‘0’.

*Methods**:***

public void initialiaze (URL url, ResourceBundle rb): As this panel implements the Initializable interface, we initialized all the images and their positions here.

3.1.6 Help Panel



Game will have a help panel to give information about gameplay and features. Player can get tips about types of enemy tanks and types of walls. Also player can be informed about rules and game controls. Player can reach the help panel from the main menu of the game.

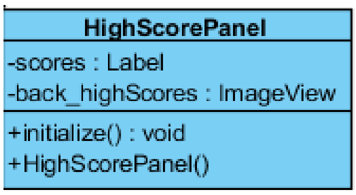
This class makes use of JavaFX's Initializable[1] class which accesses an FXML file containing all of the layout information. This will allow to add more design specifications in the way that this Menu looks like.

Also in this way we can improve on design details by making use of the FXML-s.[1]

*Methods**:***

public void on_back_help _Pressed(ActionEvent e): This method describes the action which will be performed after the back button is pressed. It describes the way to get back to the main menu.

3.1.7 HighScores Panel



When high scores button is pressed, system will show the list of top ten scores with player names. If player can make a score which is higher than the 5th score, his score is displayed at this list and he will enter his name to high scores list.

This class makes use of JavaFX's Initializable[1] class which accesses an FXML file containing all of the layout information. This will allow to add more design specifications in the way that this Menu looks like.

Also in this way we can improve on design details by making use of the FXML-s.[1]

Constructor:

public HighScorePanel (): The constructor for High Scores. This constructor creates a panel which is going to hold **scores** and **back_highscores**.

Attributes:

private Label scores: this text label will hold the list of all highest scoring users ranked 1 to 5 in a column of names and corresponding points collected

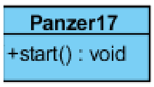
private ImageView back_highscores: this imageView keeps the image of the previous page.

Methods:

public Array getHighScore (): Reads the recorded highscore information. This method will be using the **fileManager** to access the high scores which were previously saved in a text. This method returns an array of strings, including the names and the points of the player .

public void initialize(URL url, ResourceBundle rb): As this panel implements the Initializable interface, we initialized all the images and their positions here.

3.1.8 Panzer17 Class



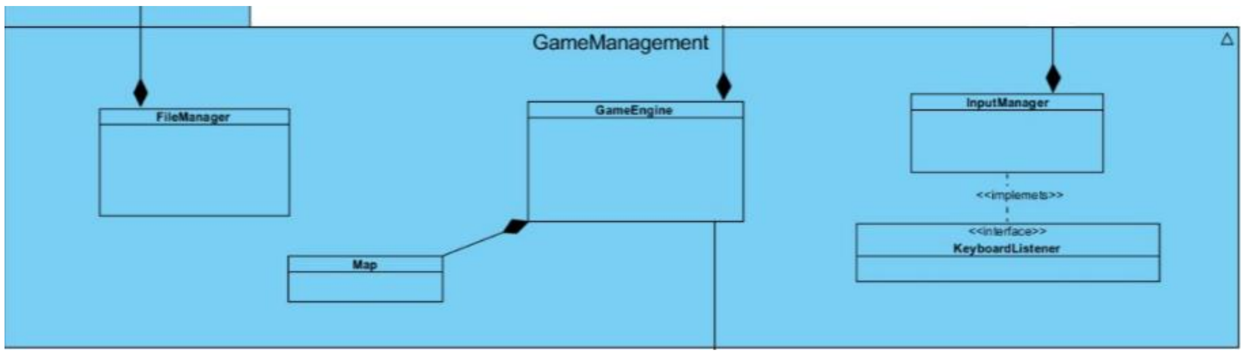
This is the main class of our project. It extends Application to make use of JavaFX's Stage and Scene features.

*Methods***:**

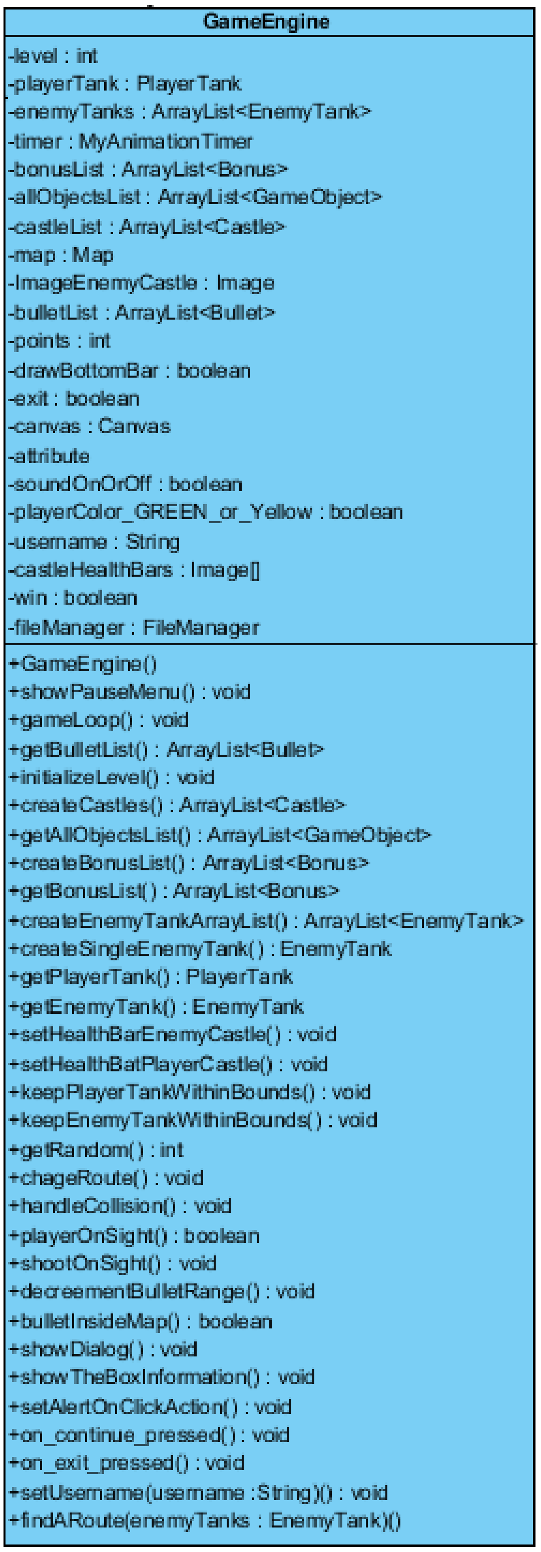
public void start(Stage stage): There is a class needed to set up the main menu so this class set sup the main menu with ‘stage’ functionalities**.**

3.2 Game Management Subsystem Interface

Game Management Subsystem consists of four managers. Game Engine is the façade class which means all the classes are controlled by this class. This subsystem is responsible for starting the game, taking the inputs from the player, updating settings and returning the text files when the highscore and credits panels are invoked. Furthermore, this class is also responsible for the Map management. Map class plays a key role which is described in details in the upcoming pages of the report.



3.2.1 GameEngine Class



```
GameEngine
-level : int
-playerTank : PlayerTank
-enemyTanks : ArrayList<EnemyTank>
-timer : MyAnimationTimer
-bonusList : ArrayList<Bonus>
-allObjectsList : ArrayList<GameObject>
-castleList : ArrayList<Castle>
-map : Map
-ImageEnemyCastle : Image
-bulletList : ArrayList<Bullet>
-points : int
-drawBottomBar : boolean
-exit : boolean
-canvas : Canvas
-attribute
-soundOnOrOff : boolean
-playerColor_GREEN_or_Yellow : boolean
-username : String
-castleHealthBars : Image[]
-win : boolean
-fileManager : FileManager
+GameEngine()
+showPauseMenu() : void
+gameLoop() : void
+getBulletList() : ArrayList<Bullet>
+initializeLevel() : void
+createCastles() : ArrayList<Castle>
+getAllObjectsList() : ArrayList<GameObject>
+createBonusList() : ArrayList<Bonus>
+getBonusList() : ArrayList<Bonus>
+createEnemyTankArrayList() : ArrayList<EnemyTank>
+createSingleEnemyTank() : EnemyTank
+getPlayerTank() : PlayerTank
+getEnemyTank() : EnemyTank
+setHealthBarEnemyCastle() : void
+setHealthBatPlayerCastle() : void
+keepPlayerTankWithinBounds() : void
+keepEnemyTankWithinBounds() : void
+getRandom() : int
+chageRoute() : void
+handleCollision() : void
+playerOnSight() : boolean
+shootOnSight() : void
+decreementBulletRange() : void
+bulletInsideMap() : boolean
+showDialog() : void
+showTheBoxInformation() : void
+setAlertOnClickAction() : void
+on_continue_pressed() : void
+on_exit_pressed() : void
+setUsername(username :String)() : void
+findARoute(enemyTanks : EnemyTank)()
```

This class is like the Façade class of the Game Management Subsystem. This class is the core class of this game where the game logic is produced and combined to perform the functionality of the game. In order to show the results this class makes use of **ObjectDrawer** which takes commands from **GameEngine** and draws the corresponding objects onto the screen.

Attributes:

-private int level: this attribute is used to know which level the player is playing currently. Player cannot choose a level so the system with the help of this attribute increments the level when the previous one is finished.

-private boolean soundOnOrOff: this attribute represents the sound setting. If set to **true** it will enable sound, and if set to **false** it will disable sound from being played throught the game. The method **playSound()** will make use of this variable when it is evoked.

-private boolean playerColor_GREEN_or_Yellow: this attribute sets the particular images to the **PlayerTank.** İf set to **true** the **PlayerTank** will be represented by green pictures of the tank and if set to **false PlayerTank** will be represented by yellow pictures of the tank. The data regarding this choice is parsed with the help of **fileManager**'s **getSettings()** method.

-private PlayerTank playerTank: this object holds the data for the current player, say the user in this case. The playerTank is initailized as a **Tank** with normal speed, full life points which amount to 5, ability to move around the map and ability to shoot **MetalBullet** or **IceBullet** depending on the **Bonus** acquired.

-private String username: this string represents the name of the user playing the game. It is initialized through a set function through a prompt dialog at the very beggining of the game. Then, if the player collects enough points his/her name will be added to the highscores by using **FileManager.**

-private ArrayList<EnemyTank> enemyTanks: this attribute represents a an array which holds all of the **EnemyTank**'s currently in the game. Through this attribute we can access each enemy and modify its properties such as **frozenState** to decide whether that enemy should be frozen or even **life** which decide whether an **EnemyTank** is not alive and should be deleted from the array.

-private ArrayList<Bonus> bonusList: this attribute represents an array which holds a list of **Bonus** objects. The generation of the bonuses is done in a random fashion through **createBonusList()** which initializes different types of bonuses and stores them in this array for future use.

-private ArrayList<Castle> castleList: this attribute holds the castles present in the game namely **PlayerCastle** and **EnemyCastle**

-private ArrayList<Bullet> bulletList: this attribute holds all the bullets present at a certain instant on the game. We need a collection of the bullets so that we can decide about the speed at which bullets will move onto the map. This array is used by **decrementBulletRange()** to implement bullets lifespan.

-private ArrayList<GameObject> allObjectsList: this array represents all the objects present at a particular moment within the game. It is crucial to the game in that we use it to draw all of the objects in the game **Brick**, **Tank**, **Bonus**,**Bullet, Castle.** A delete or addition within this array would be directly detected from **ObjectDrawer** which keeps refreshing the screen at 60 fps by drawing all of the objects in this arraylist to the screen. Say an **EnemyTank** was shot and life points decresed to 0, he is declared dead and deleted from this arraylist, meaning **ObjectDrawer** will not draw it anymore onto the map in the next frame thus kicking that object out of the game.

-private Map map: this attribute makes use of the **Map** and gives us acces to all of the **Brick**s of the game as well as their postion in the map and life points. **Map** makes sure it initializes the **Brick**s according to the current level of the game by accessing the level data from text files.

-private Image[] castleHealthBars: this attribute represents the images used to view the health bar. **ObjectDrawer** uses these images to reflect the life points of the **Castle**s by drawing the appropriate image onto the bottom bar.

-private int points: this attribute keeps hold of the current points the player has at a particular moment. The incrementation is done in the **handleCollision(..)** method when player's **Bullet** hit another object, be it **Brick, EnemyCastle** or **EnemyTank.**

-private boolean drawBottomBar: this attribute will be used to decide whether status bar at the bottom of the screen needs to be redrawn or not. The **Castle** health bars, **Bonus** bar (representing current Bonus collected by drawing bonus image) are built through drawing pictures but drawing the same picture 60fps would overload the **ObjectDrawer** and cause it to become unresponsive. To help solve that problem we only redraw the pictures when they need to, let's say **Castle** health bar images only need to be drawn when **handleCollision(..)** detects a **Castle** being hit by a **Bullet** only.

-private static boolean exit: this attribute is used to manage game exit scenarios. When **handleCollision(..)** detects **PlayerTank** or **PlayerCastle** life points went to 0 it evokes **showDialogAndAct(…)** which checks the state of this boolean and decides whether exit from the game is needed or not, say in case the player lost the game.

-private Canvas canvas: this attribute holds a reference to the current canvas onto which the **ObjectDrawer** is continually drawing. Using this reference we can get the current **Scene[1]**and access the **Stage**(reference here) that we are in which is needed to evoke **showPauseMenu().**

-private boolean win: this attribute is used to assist in deciding whether the player has won the game or not. Through this attribute we can decide whether to restart the game. It is used by **showDialogAndAct(…)**

-private FileManager fileManager: this attribute represents an instance of the **FileManager** class which is used on **showDialogAndAct(…).** In case the user has collected enough points, this method will use **fileManager** to write the score into the highscores list.

-private ObjectDrawer objectDrawing: this attribute represents an instance of the **ObjectDrawer** class which is used to access objects instantiated within the **GameEngine** and regulate the timing onto which the objects are drawn onto the map.

Constructors:

public GameEngine(): it initializes the **level** to 1 and creates an instance of the **fileManager** which will be used to save highscores at the end of the game. It also initializes settings of the game such as **soundOnOrOff** and **boolean playerColor_GREEN_or_Yellow** by getting the saved settings from the txt through **fileManager'**s **getSettings()** method.

Methods:

-public void setUsername(String username): this method **sets** the username attribute. It will be used when user starts the game

-public void gameLoop(): this method initializes all of the objects of the game by putting them in separate ArrayLists: **enemyTankList, bonusList, castleList** by making use of **createCastles(), createBonusList(), createEnemyTankList().** İt also initiates **bulletList** (which will be filled in later during gameplay), as well as the **playerTank** with its initial attributes**.** It then puts all of this objects into **allObjectsList** to have a record of all objects in the game. Except from these it also initializes the **castleHealthBars** with the full health bar images and all these being done it evokes the **ObjectDrawer** which will then get access to all of these lists and start drawing them on the screen.

-public ArrayList<Castle> createCastles(): this method initializes both **PlayerCastle** and **EnemyCastle,** puts them into an arraylist and then returns it.

-public ArrayList<Bonus> createBonusList(): this method initializes an arraylist of **Bonus** objects depending on the current **level** and then returns it.

-public ArrayList<EnemyTank> createEnemyTankList(): this method returns an arraylist of different types of **EnemyTank**'s depending on the current **level.**

-public ArrayList<GameObject> getAllObjectsList(): this method functions as a **get** method to return **allObjectsList.**

-public ArrayList<Bullet> getBulletList(): this method **returns** the **bulletList.** İt will be used by **EnemyTank** and **PlayerTank'**s to add **Bullet**'s into the **bulletList** when the tank shoots a bullet.

-public PlayerTank getPlayerTank(); this method returns **playerTank.** Sınce the latter is declared private we can only access it through this method instead.

-public PlayerTank setHealthBarEnemyCastle (Image img) : this method gets an Image file as a parameter and sets the corresponding **castleHealthBars** in this case the **EnemyCastle** health bar to another picture if that castle was hit by a bullet. In that case the newly set picture will depict that the **EnemyCastle** has lost life points.

-public PlayerTank setHealthBarPlayerCastle (Image img) : this method gets an Image file as a parameter and sets the corresponding **castleHealthBars** in this case the **PlayerCastle** health bar to another picture if that castle was hit by a bullet. In that case the newly set picture will depict that the **PlayerCastle** has lost life points.

-public void showPauseMenu(): this method will use **canvas** to get the context and the current **Stage** (reference here) and assign the new view to the **PauseMenu.** This method being evoked does not get rid of the past **Scene** of the gameplay but just shows the **PauseMenu Scene** on top of the gameplay scene. This method also makes sure that the gameplay scenery is paused by calling **stop()** on the **ObjectDrawer.**

-public void keepPlayerTankWithinBounds() : This method makes sure the **playerTank** is incapable of moving if he attempts to move out of the map. It is evoked in the **ObjectDrawer** which means that it is being called nearly 60 times per second.Thus the method should be able to access the **playerTank** and get its coordinates. İf the x and y coordinates of the **playerTank** result to be out of the box of the game (which includes the part of the canvas where GameObjects are drawsn) then the player speed will be set to 0 thus disabling the **playerTank** to go out of the map. But if the player then moves in the opposite direction the speed will be set to normal back again and the **playerTank** moves. So as the method name goes the **playerTank** is kept within boundaries of the gamemap.

-public void keepEnemyTankWithinBounds (EnemyTank tank) : This method resembles **keepEnemyTankWithinBounds()** but it has to be in a separate method since the **EnemyTank** movement is not set through the keyboard presses but through method calls, specifically it is **findARoute()** which first starts the random tank movement. In particular this method detects whether the **EnemyTank'**s coordinates are out of the map boundaries. If the boundary coordinates match with the tanks current coordinates the method invokes the **move** functions(**moveUp**(), **moveDown**(), **moveLeft**(), **moveRight**()) to change the direction of the tank. In this way the **EnemyTank** always makes his random movement within the game map boundaries.

-public int getRandom(): This method returns a random integer

-public void findARoute(EnemyTank enemyTank): This method first checks if the parameter's **EnemyTank** is frozen or not, a case in which the enemy should not be able to move. If the **EnemyTank** is not frozen than the method generats a random number and uses this random number. This random number is then used in 5 if statements which all correspond to different range of the random number. To increase randomness of the tank movement the if statements check the current direction of the tank and set it to move in another direction. Ex: If the direction of the tank is **UP** depending on the if statements the tank will either move **DOWN, LEFT** or **RIGHT.**

public boolean playerOnSight(EnemyTank shooterTank): This method checks whether the parameter's **EnemyTank** is within a certain range near to the player tank.It also calculates whether if **EnemyTank** shoots, the **Bullet** will aim at the body of the **PlayerTank.** This will add difficulty to the game in that the user has to move his tank fast and has little chance of avoiding enemy fire. If the **PlayerTank** is within this range then the methods return **true** otherwise it returns false.

public boolean shootAndMove(EnemyTank shooterTank): This method implements what happens when **EnemyTank** shoots the **playerTank**. It first detects whether the **playerTank** is within reach. Then if that results to be true it will shoot towards the tank. In case the **playerTank** moves away it will follow him by getting his direction and moving correspondingly. Also, if **playerTank** stops than the **EnemyTank** will keep shooting but stop 5 pixels in front since we do not want them to collide with each other.

public boolean decerementBulletRange (): This method accesses the **bulletList** decrements their range if it was previosuly incremented. When it is shot, the **Bulley** initially has a certain range > 0. If the range is > 0 this method will decrement the **Bullet** range by 15. The method does this once for each bullet that has a range bigger than 0 yet since we run this method in the **ObjectDrawer** the same operation will be called 60 times per second. Thus this decrement will therefore appear as the actual bullet shot. It also limits the distance at which the **Bullet** can shot at since shooting a **Bullet** from one side of the map to the other end would end the game very fast. Instead we implemented that as Bonus feature. For more check **FastBulletBonus.**

public void showDialogAndAct (boolean won, int level): This method is evoked on critical end points such as when the player enters a new level, clear all levels or loses the game. It makes use of the parameters and shows a dialog box in each of these three different cases as well as the action that comes with it. In case level is cleared it reinitializes the game to the next level. If all levels are cleared or game is lost(EnemyCastle life points = 0), it returns to the MainMenu.

public void handleCollision (): This method handles all of the collision that happen throught the game. It makes comparisons between all objects by using nested loops. The possible scenarios to be considered are :

- **PlayerTank** collides with **Brick :** In this case we set the speed of the **PlayerTank** to 0.
- **EnemyTank** collides with **Brick :** In this case it moves the **EnemyTank** a pixel in the opposite direction to avoid merging with the **Brick** and than calls **findARoute(...)** on that tank such that when avoiding the **Brick** it will not merge with another **Brick.**
- **EnemyTank** collides with **PlayerTank :** You would not expect that to happen but either way you we set their speeds to 0 when they try to move closer to each other than they should. In this case we change their coordinates in the respective opposite direction as well
- **EnemyTank** collides with **EnemyTank :** In this case both of the enemies tank move in different directions by the evoking of **findARoute(...)** on each of them
- **Bullet** collides with **Brick** : In this case, if the **Bullet** owner is a **PlayerTank** the Brick's life points are decremented by 1 and the **PlayerTank** gains one point. The same thing happens if the Bullet belongs to an **EnemyTank** with the sole difference that no points are added to them.
- **Bullet** collides with **EnemyTank :** In this case we need to avoid friendly fire so if the bullet object belongs to an **EnemyTank** nothing happens but if the bullet belongs to the **PlayerTank** than there are **two** cases:
  - **1. Bullet** is an instance of **MetalBullet : 1** life point is decremented to the **EnemyTank** affected. If **EnemyTank**'s life points are decremented to 0, it will be removed from the **allObjectsList** thus it will not be drawn in the next frame onto the canvas by the **ObjectDrawer** in this way we implement a real enemy dead scenario. In order to make it look like a collision **Bullet** involved is stopped and deleted as well at the first instant of the collision. Playing a small explosion animation here is something we consider as a non-functional requiremnt for the time being.
  - **2. Bullet** is an instance of **IceBullet:** The speed of the **EnemyTank** is set to 0,its icon is set to one that resembles an ice-frozen tank and we also call

incrementFrozenStateDuration() and set the tanks **frozenState** to **true** by calling **setFrozenState**() on the tank. This will give a real feel to the game but it is something that deffinitely has to be reverted. So after some time the **EnemyTank** will have its normal icon and be able to move again say after 7 seconds. It is the **ObjectDrawer** that checks the **frozenStateDuration** and decrements it at each frame of a second. When **frozenStateDuration** is 0 the enemy's speed will be set to normal as well as its icon, thus enabling it to move.

- **Bullet** collides with **PlayerTank :** Here as well there are two cases:
  - **1. Bullet** is an instance of **IceBullet** : If the **PlayerTank frozenState** is false (you only freeze the tank if its un-frost) evoke **PlayerTank incrementFrozenStateDuration()** and set its frozen state to true by evoking **setFrozenState**(...) on the tank object. The unfreezing procedure is the same with the one **EnemyTank**'s in that we change the image, set speed to 0 and revert after several seconds to normal. İn this period **PlayerTank** is not affected by other bullets.
  - **2**. **Bullet** is an instance of **MetalBullet:** İf the **PlayerTank** has 0 life points he is removed from **allObjectsList** thus it will not be drawn in the next frame onto the canvas by the **ObjectDrawer.** In that case **showDialogAndAct(...)** is evoked, thus getting the **win** as false. In that case game is declared lost and user is returned to **MainMenu.** If **PlayerTank** has more than 0 life points, its life points are decremented. To show the life points, a lifepoint decrement also changes the images assigned to the **PlayerTank** in that a small healthbar is shown on top of the Tank
- **Bullet** collides with **EnemyCastle :** Firstly **Bullet** is deleted from **allObjectsList .** In this case we decrement **EnemyCastle**'s life points. To show its life points we change the image of the health bar using **setHealthBarEnemyCastle().** If life points are 0, game is declared **won,** and **showDialogAndAct(...)** is evoked. The method will

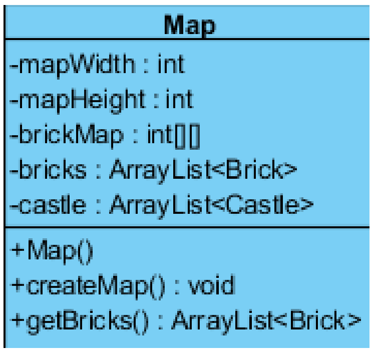
decide whether it is a level cleared scenario or a game cleared scenario and act acordingly.

- **Bullet** collides with **PlayerCastle :** Firstly **Bullet** is deleted from **allObjectsList .** Then if the **Bullet** owner is not a **PlayerTank,** than **PlayerCastle** gets decremented 1 life points. To show its life points we change the image of the health bar using **setHealthBarPlayerCastle().** If life points are 0, game is declared **lost,** and **showDialogAndAct(...)** is evoked. The method will decide will then return the user back to **MainMenu.**
- **Bonus** collides with **PlayerTank:** There are different **Bonus'** thus we have different cases:
  - **Bonus** is instance of **SpeedBonus**: İn this case we set **PlayerTank's** speed to 3. (normal is 1), set the time bonus is used for by calling **incrementMyBonusDuration()** on the **PlayerTank,** destroy the **Bonus** by calling **setBrute_destroy(...)** onto it**.** Also we set **drawBottomBar** in order to allow **ObjectDrawer** to draw a picture of the Bonus at the bottom bar of the screen showing the duration as well.
  - **Bonus** is instance of **ProtectionBonus**: İn this case we set **PlayerTank's hasShieldProtection** to true by using **setShieldProtection()** and also increment this tanks bonus duration (how long this bonus is valid for) by using **incrementMyBonusDuration().** We also destroy the **Bonus** by calling **setBrute_destroy(...)** onto i**.** Also we set **drawBottomBar** in order to allow **ObjectDrawer** to draw a picture of the Bonus at the bottom bar of the screen showing the duration as well.
  - **Bonus** is instance of **EnemyFreezeBonus**: İn this case we evoke **setHasIceBullet()** to **PlayerTank** which will enable it to shoot with **IceBullet's .**Also we increment this tanks bonus duration (how long this bonus is valid for) by using **incrementMyBonusDuration().** We also destroy the **Bonus** by calling **setBrute_destroy(...)** onto it**.** Also we set **drawBottomBar**

in order to allow **ObjectDrawer** to draw a picture of the Bonus at the bottom bar of the screen showing the duration as well.

- **Bonus** is instance of **FullHealthBonus**: İn this case we evoke set the life points of both **PlayerTank** and **PlayerCastle** to full and change their corresponding health bar images as well. Bonus is destroyed instantly on collision by by calling **setBrute_destroy(...)** onto it.
- **Bonus** is instance of **FastBulletBonus**: İn this case we evoke **setHasSuperBullet ()** to **PlayerTank** which will enable it to shoot **MetalBullets** faster and with a higher range**.**Also we increment this tanks bonus duration (how long this bonus is valid for) by using **incrementMyBonusDuration().** We also destroy the **Bonus** by calling **setBrute_destroy(...)** onto it**.** Also we set **drawBottomBar** in order to allow **ObjectDrawer** to draw a picture of the Bonus at the bottom bar of the screen showing the duration as well.
- **Bonus** is instance of **CoinsBonus**: İn this case we increment the points of the **PlayerTank** depending on this **Bonus'es bonusPoints** and then delete this Bonus from the map by calling **setBrute_destroy(...)** onto it.

3.2.2 Map Class


Map
-mapWidth : int
-mapHeight : int
-brickMap : int[][]
-bricks : ArrayList<Brick>
-castle : ArrayList<Castle>
+Map()
+createMap() : void
+getBricks() : ArrayList<Brick>

This class is responsible for the full screen maps. The maps will change according to the levels, tanks, bricks and the coordinates of the tanks. This map plays a key role in user interface because it should update the map according to the needs of the system. This map does not change only when player presses ‘Pause’.

Attributes:

private int mapWidth: this attribute is needed because this shows the screen width and according to the width and height attributes, the map fits to the screen.

private int mapHeight: this attribute is needed because this shows the screen width and according to the width and height attributes, the map fits to the screen.

private int[][] brickMap: this attribute plays a key role in the game. According to the number of bricks collapsed, the Map class will automatically update itself and it updates itself with the help of this brickArray. This 2D array shows where the bricks value is 0 which means the brick is gone and 1 which means the brick exists.

private ArrayList<Brick> bricks: as it can be seen from the name of the attribute, this attribute references to the Castle object. This attribute holds an arraylist of brick objects.These bricks will be seen in the Map.

private ArrayList<Castle> castle: as it can be seen from the name of the attribute, this attribute references to the Castle object. This attribute holds an arraylist of castle objects. These castles will be seen in the Map.

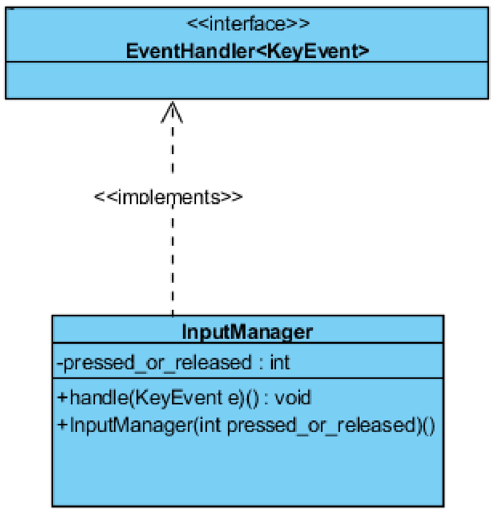
Constructors:

public Map (int mapWidth, int mapHeight, int level): initializes a Map object with default attribute values. When invoked, this map constructor will update a map according to its level. Therefore, we defined mapWidth, mapHeight and level as parameters.

Methods:

public void createMap(int row,int col, int level): creates the current map according to the given attributes.

3.2.3 InputManager Class



<<interface>>
EventHandler<KeyEvent>
<<implements>>
InputManager
-pressed_or_released : int
+handle(KeyEvent e)() : void
+InputManager(int pressed_or_released)()

This class implements **EventHandler**<KeyEvent> in that it will decide what happens when user presses the controls from the keyboard.

Constructor:

public InputManager (int pressed_or_released): The constructor decides whether the **InputManager** is being used for the event when the button is released or when the button is pressed. Thus if a button press would move the **playerTank** the button release (which will eventually come right after) would stop the tank movement accordingly thus preventing it from moving infinitely. So this class can be used eithe as an **OnKeyPressed** action handler or a **OnKeyReleased** handler. The distiguishment is done through the parameter that it receives.

Attributes:

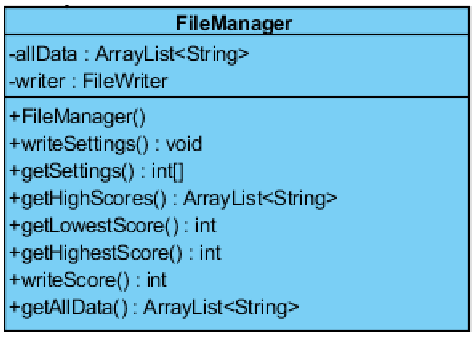
private int pressed_or_released: This attribute decides whether the handler will consider the event as an **OnKeyPressed** or an **OnKeyReleased.** İt is set through the constructor by assigning it the value of the parameter.

Methods:

public void handle(KeyEvent e) – This is an “overrided” method which is intended to implement **EventHandler<KeyEvent>**.

- In case **pressed_or_released** = 1, it means that our caller wanted to call for an **OnKeyPressed** event and thus we enter the first if conditional. Within this conditional we use a switch statement to decide what happens when we press arrow keys. For each movement it calls **moveInDirection(arg1, arg2)** onto the **PlayerTank** with **arg1** being **PlayerTank's** current direction and **arg2** being **true** (true means coordinates of the **PlayerTank** will be changed which is instantly reflected on the **ObjectDrawer** which draws it in the new coordinates, thus giving the perception of motion). We also evoke **shootMetal(...)** when we detect **SPACE** was pressed and **shootIce(...)** when **B** is pressed from the keyboard having that the **PlayerTank** has the ability to shoot **IceBullets** (**hasIceBullet()** for the **PlayerTank** should return true).
- In case **pressed_or_released** = 2, it means that our caller wanted to call for an **OnKeyReleased** event and thus we enter the second conditional block. Within this conditional we use a switch statement to decide what happens when we press arrow keys. For each movement it calls **moveInDirection(arg1, arg2)** onto the **PlayerTank** with **arg1** being **PlayerTank's** current direction and **arg2** being **false** (false means coordinates of the **PlayerTank** will **not** be changed by setting the **PlayerTank**'s speed to 0 which will cause the object to stop; this is indeed reflected on the **ObjectDrawer** as well since it will draw the **PlayerTank** at the same location thus making it stop).Additionally we also put the "**P"** button in our switch case and call **showPauseMenu()** to show the PauseMenu on the screen. In the last case we also need to pause the game thus we call **stop()** onto the **ObjectDrawer.**
-

3.2.4 FileManager Class



This FileManager class controls the text files and returns the text files according to the player. Therefore, this class also interacts with the User Interface Subsystem. For example, when the player presses 'High Score' button, User Interface System sends a command to the File Manager to take the file and getList method returns the array of strings.

Attributes:

private ArrayList<String> allData: it is arraylist of username and highscores where every element is represented by a row.

private FileWriter writer: this attribute is needed when in writeSettings, the update to the settings should be done. For example: writer = new FileWriter("src/panzer/pkg2017/files/saved_data.txt", false);

Constructors:

public FileManager(): initializes the attributes of this class. Since every level has its own map, these maps should be initialized in the first run of the system via this constructor.

Methods:

public void writeSettings(int sound, int playerColor) : with this method, if the player changed any of the settings, it will be updated into a .txt file with this method.

public int[] getSettings(): this method returns an array of two integers, the first index represents the sound settings (on or off= 1 or 0) while the second index represents the player color setting (green or yellow = 1 or 0)

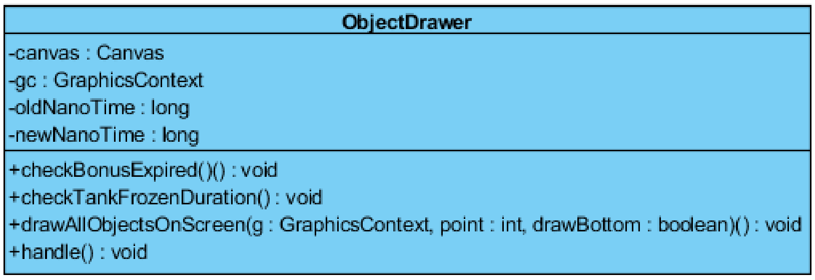
public int getLowestScore(): this method returns the lowest score from the .txt file

public int getHighestScore():this method returns the highest score from the .txt file

public int writeScore(): this method will update the highScores .txt file with writing the score of the player within username.

private ArrayList <String> getAllData(): returns an arraylist of elements , each element represent a row. For example :first row = settings {"01"}, other rows = highscore = {john_12540}

3.2.5 ObjectDrawer Class



```
ObjectDrawer
-canvas : Canvas
-gc : GraphicsContext
-oldNanoTime : long
-newNanoTime : long
+checkBonusExpired()() : void
+checkTankFrozenDuration() : void
+drawAllObjectsOnScreen(g : GraphicsContext, point : int, drawBottom : boolean)() : void
+handle() : void
```

This class implements all of the drawing that happens on the screen. It extends **AnimationTimer** from JavaFX. In order to use it we need to override **handle(…)** method. This method will be called an average of 60 times per second depending on machine. This means that we will be drawing our objects at 60fps and in order to implement the motion of objects we change their coordinate attribute. These coordinates are given to the **draw(…)** function as parameters to draw the object on the canvas. Since our GameObject all have **Image** attribute we will use specifically **drawImage().**Details are to be explained within the information below.

*Attributes**:***

private Canvas thisCanvas: This attribute represents the canvas onto which we will be drawing. All of the figures will be drawn onto this canvas.

private GraphicsContext gc : We make use of this attribute in order to use the **Canvas**' **draw(...)** methods.

private long oldNanoTime: This attribute holds the time in milliseconds. It is reassigned on each frame through the **System.nanoTime();**

private long newNanoTime: This attribute is designed to hold the new time in millisecond. So if the difference between **oldNanoTime** and **newNanoTime** reaches 1000 milliseconds , it means one second has passed. In this way we design the timing of our **AnimationTimer.**

*Methods**:***

-public void checkBonusExpired() – Thıs method checks whether the **PlayerTank** still holds the **Bonus** capabilities or not. So if **PlayerTank**.**getMyBonusDuration()** returns true, it means the **PlayerTank** has collected a **Bonus** and is using it. In this case we call **decrementMyBonusDuration()** onto the **PlayerTank.** Since **checkBonusExpired()** will be run onto the **handle(...)** method it means it will call **decrementMyBonusDuration()** at a rate of 60fps. When **myBonusDuration** gets 0 the player should revert all of its extra powers he gained through the **Bonus'es**

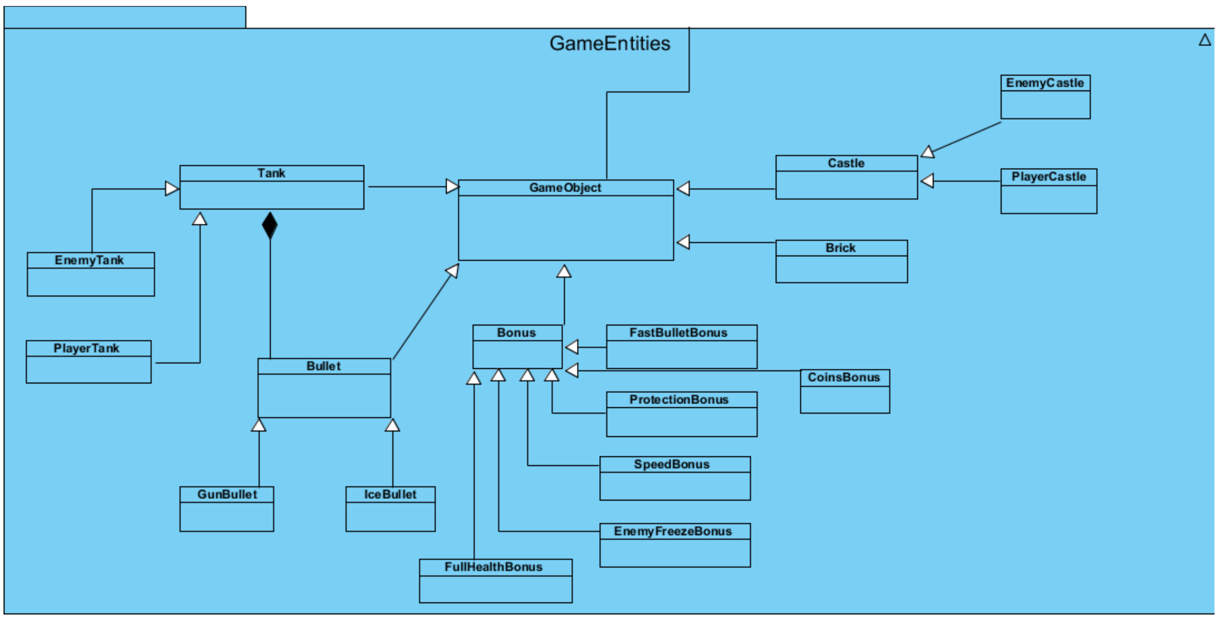
-public void checkTankFrozenDuration() – This method checks if the **PlayerTank** or **EnemyTank'**s getFrozenStateDuration() returns a value bigger than 0. In that case it means frozen state time has elapsed and the duration should be decremented. To decrement this value we call **decrementFrozenStateDuration()** on the corresponding **Tank. checkTankFrozenDuration()** will be called onto the **handle(...)** method it means it will call **decrementFrozenStateDuration ()** at a rate of 60fps. When the decrement reaches 0. It will set the corresponding **Tank** frozen state to false thorugh **setFrozenState(false)** and reset the picture from a picture with ice to the normal one through **setCustomImg(image).** In this way we implement a temporary player freezing.

-public void drawAllObjectsOnScreen(GraphicsContext g,int point,int time, boolean drawBottom) : this method accesses **allObjectList.** It runs a loop onto each of its **GameObjects** and calls **g.drawImage(image, x, y)** by accessing each **GameObjects** image, as well as its current x and y coordinate attribute. The method will draw the image onto the canvas which will then be visible to the user.If we deleted a GameObject from the **handleCollision()** it means it will not be in **allObjectList** anymore, thus it will not be drawn onto the **Canvas** as well.This method also is responsible for drawing the bottom bar health images as well.

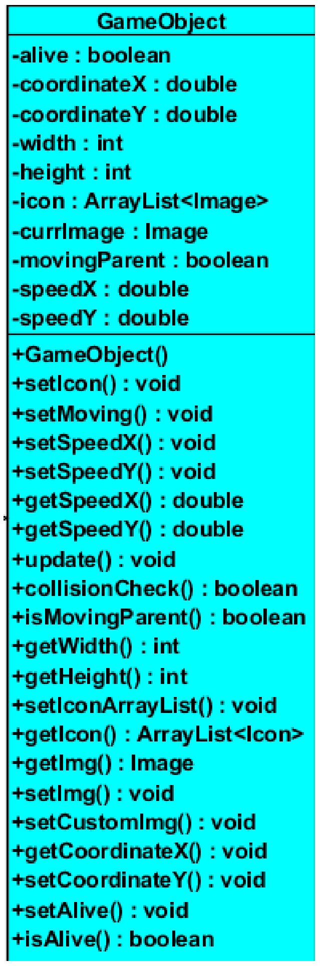
- public void handle(long now) – This method is overrided from **AnimationTimer.** When we call **start()** to the **ObjectDrawer** this method will go rerun itself in a rate of 60 fps. It will only stop if we call **stop().** İn order to implement all of the movement and drawing onto the canvas we make use of the **now** attribute which returns the current time at which method is being called in milliseconds. Within the method we call **checkTankFrozenDuration(),checkBonusExpired()** and **drawAllObjectsOnScren(...).** Calling them here at 60fps will give our game a smooth look and feel.

3.3 Game Entities Subsystem Interface

Game Entities is a subsystem that is composed of the main graphic objects of the game. The whole system is dependent on the **MapBuilder** class which initiates the map of the game for the first time thus drawing all of the objects such as **Tank**s, **Brick**s and **Castle**s onto the map. After the map is created the game starts and the user or enemies may initiate **Bullet**s toward each other. When enemy tanks are dead a **Bonus** will appear at the place of death and the **PlayerTank** can collect it to upgrade his/her tank. Below we show a diagram of this subsystem and the detailed description of each class separately.



3.3.1 GameObject Class



```
GameObject
-alive : boolean
-coordinateX : double
-coordinateY : double
-width : int
-height : int
-icon : ArrayList<Image>
-currImage : Image
-movingParent : boolean
-speedX : double
-speedY : double
+GameObject()
+setIcon() : void
+setMoving() : void
+setSpeedX() : void
+setSpeedY() : void
+getSpeedX() : double
+getSpeedY() : double
+update() : void
+collisionCheck() : boolean
+isMovingParent() : boolean
+getWidth() : int
+getHeight() : int
+setIconArrayList() : void
+getIcon() : ArrayList<Icon>
+getImg() : Image
+setImg() : void
+setCustomImg() : void
+getCoordinateX() : void
+setCoordinateY() : void
+setAlive() : void
+isAlive() : boolean
```

This class will be the base for most of the other entities. It specifies the location of the object, its speed as well as a **square** image icon assigned to the object. A **GameObject** may be dead or not. As the objects they might collide with each other, meaning that their respective x and y coordinates will match (inside a square perimeter). This collision is caught and the type of collision is then specified, whether a tank hits another tank, a castle or a brick.

Constructor:

public GameObject(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height)

This constructor initiates a basic **GameObject.** This object will primarily have a position (x,y) in the map as well as a health state which determines if the GameObject is alive or not.It also initialize width and height of the object on the panel. All of the **GameObjects** can be destroyed with no exception.

Attributes:

private boolean alive: this Boolean value specifies if the GameObject lives or not. If **alive** is set to false it means that the object has either lost all of the life points or expired and now needs to be removed off the map.

private double coordinateX: specifies the upper left corner X coordinate of which the GameObject will be placed.

private double coordinateY: specifies the upper left corner Y coordinate of which the GameObject will be placed.(We need the upper left corner since the draw methods need the upper left corner coordinates to draw the image over the map)

protected double speedX: specifies the speed under which a **GameObject** may move in the horizontal X direction**.** It applies only for the **Tank** and **Bullet** class however since the other objects will not be moving.

protected double speedY: specifies the speed under which a **GameObject** may move in the vertical Y direction**.** It applies only for the **Tank** and **Bullet** class however since the other objects will not be moving.

private ArrayList<Image> icon: specifies the images which is attributed to the **GameObject.** It is needed to distinguish between different types of **GameObjects** and it is accessed from previously designed icons in the local storage.

private int width: specifies the width of the object on the map. Accorging to this width value, drawing functions in the GameEngine class will draw the objects on the panel.

private int height: specifies the height of the object on the map. Accorging to this height value, drawing functions in the GameEngine class will draw the objects on the panel.

private Image currImage: Current image of the GameObject is hold on this variable to draw the object in the correct direction with the correct image.

Methods:

public void setIcon(ArrayList<Image> icon): According to draw the **GameObject**, **ArrayList of Images** will be used. **setIcon** function takes and array list of image as parameter and assign it to the icon attribute of the object.

public void setSpeedX(double speedX): This function takes a double input and assign it to the **speedX** attribute of the **GameObject**. This speed attribute will be used in update function to change the coordinates of the object.

public void setSpeedY(double speedY): This function takes a double input and assign it to the **speedY** attribute of the **GameObject**. This speed attribute will be used in update function to change the coordinates of the object.

public double getSpeedY(): Returns the **speedY** attribute of the object. Returned value is used to draw the object and move the object.

public double getSpeedX(): Returns the **speedX** attribute of the object. Returned value is used to draw the object and move the object.

public void update(int endOfMapX): Update function controls the coordinate of the **GameObjects** to gurantee that objects are within the bounds of **GamePanel**. According to bound that objects is reached, function changes the **coordinates** and the **speed** attribute of the objects.

public boolean collisionCheck(GameObject obj): Funtion checks the collision between **Bricks,Castles,Player Tank and Enemy Tanks**. According to case, function changes the coordinates and the speed of the objects. Function returns a boolean value that shows the collision.

public int getWidth(): Function returns the **width** attribute of any **GameObject.**

public int getHeight(): Funcition returns the **height** attribute of any **GameObject**

public ArrayList<Image> getIcon(): Funtion returns the **ArrayList of Image** which is used to draw **GameObjects**

public void setIconArrayList(ArrayList<Image> img): Function changes **the ArrayList of Image** to change the apparence of the objects on the **GamePanel**. **Tank** objects have four Imges which shows the apperances in four different direction.

public Image getImg(): Function returns the current image of the object to draw the object in the **GameEngine** class.

public void setImg (int num): Function is used to select a specific image in the **ArrayList of Images** which is called as currImage attribute.

public void setCustomImg (Image img): Function is used to change the **currImage** attribute of the object.

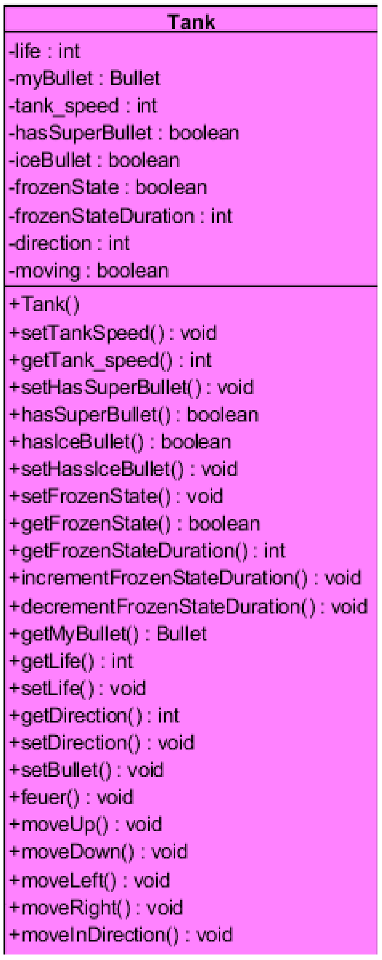
public double getCoordinateX() , public double getCoordinateY(): The two function returns the **coordinateX** and **coordinateY** attributes of any **GameObject**. Coordinates is used in **GameEngine** to control the movement and collision of objects.

public void setCoordinateX(double coordinateX), public void setCoordinateY(double coordinateY): Two setter function is used to change the **coordinate** and **coordinateY** attributes of the objects.

public void setAlive(boolean alive) : Funtion **setAlive** is used to change the **alive** attribute of objects.Alive attribute is used in **GameEngine** to add objects to **allObjectsList** and remove objects when the collision checker mechanism detects any collision which destroy objects.

public boolean isAlive(): Funtion returns the alive attribute of objects for **GameEngine** to draw them.

3.3.2 Tank Class



This class represents a Tank object. This object holds the default properties of a tank. It has **life** points, a **position**, and can also **shoot bullets** and move around the map as well. The tank “lives” as long as its **life** attribute are not zero.

Constructors:

public Tank(boolean _isAlive, float _coordinateX, float _coordinateY,int width, int height, int life):

Constructor initiates a tank which has life and isAlive properties at a certain specific location on the map (X,Y) and dimension. When created a tank must have its position and life and “_isAlive” set to true.

Attributes:

private int life : this attribute represents the current health of the tank in a specific range according to the tanks health type. If this value is zero the tank is declared as dead and is removed from the game map.

private Bullet myBullet : this attribute represents the bullet that belongs to this tank. Each tank can shoot a bullet and the user is rewarded when his/her bullet shoots another enemy. This is why each tank must have its own bullet object.

int tank_speed: Attribute holds the data for the speed of the Tank. This speed attribute is used in GameEngine to move **Tank** objects by updating the coordinates of the objects.

boolean hasSuperBullet: Attribute holds the data that whether the **Tank** object has the super bullet bonus or not.

boolean iceBullet: Attribute holds the data that whether the **Tank** object has the ice bullet bonus or not. According to truth value of this attribute shooting algorithms choose right **Bullet** to fire.

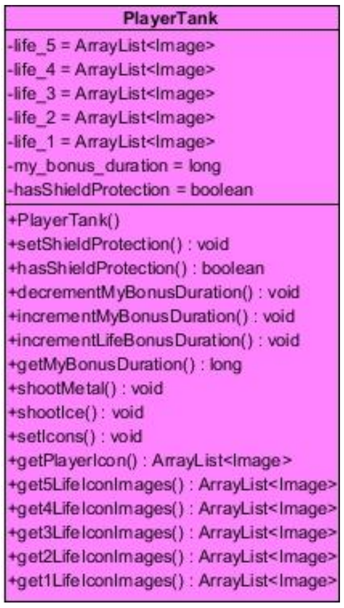
private boolean frozenState: Attribute holds the data that wheter the **Tank** object is in frozen state or not. Truth value of this attribute affects the movement of the object. When it is true, **Tank** become frozen and can’t move.

private int frozenStateDuration: Attribute holds the **time** value as an integer variable to control the duration of the frozen state.

public int direction: Direction data is hold in this attribute as an integer variable. This attribute is used to choose the right image of the **Tank** objects in four different directions.

private boolean moving: Attribute shows whether **Tank** objects are moving or not.

3.3.3 PlayerTank Class


PlayerTank
-life_5 = ArrayList<Image>
-life_4 = ArrayList<Image>
-life_3 = ArrayList<Image>
-life_2 = ArrayList<Image>
-life_1 = ArrayList<Image>
-my_bonus_duration = long
-hasShieldProtection = boolean
+PlayerTank()
+setShieldProtection() : void
+hasShieldProtection() : boolean
+decrementMyBonusDuration() : void
+incrementMyBonusDuration() : void
+incrementLifeBonusDuration() : void
+getMyBonusDuration() : long
+shootMetal() : void
+shootIce() : void
+setIcons() : void
+getPlayerIcon() : ArrayList<Image>
+get5LifeIconImages() : ArrayList<Image>
+get4LifeIconImages() : ArrayList<Image>
+get3LifeIconImages() : ArrayList<Image>
+get2LifeIconImages() : ArrayList<Image>
+get1LifeIconImages() : ArrayList<Image>

A **PlayerTank** is an extension of the **Tank,** so this class inherits all of the basic functionalities of the **Tank** class. It is the player which is commanded by the user. It has extra specific attributes such as specific type of health bar and it should support addition of **Bonus** upgrades onto it such as **shields**.

Constructor:

public PlayerTank(boolean _isAlive, float _coordinateX, float _coordinateY, int width, int height,   int life):

This constructor initiates a new Player Tank that has a certain position and dimensions on the map and full life points which may change according to collected bonuses. Initially the constructor has no bonuses, no injury and a normal speed.

Attributes:

private ArrayList<Image> life_5: PlayerTank objects has a life bar on it, according to current value of the attribute of the **GameObject**, **healthBar** on the **PlayerTank** will be changed. For the 5 state of the **healthBar**, there are 5 different **ArrayList of Images**. life_5, life_4, life_3, life_2 and life_1 attributes of the PlayerTank hold these ArrayLists of Images. Other 4 implementetain attributes of life are below.

private ArrayList<Image> life_4;

private ArrayList<Image> life_3;

private ArrayList<Image> life_2;

private ArrayList<Image> life_1;

private long my_bonus_duration: **my_bonus_duration** attribute holds the **time** value as a **long** variable to control the effect of the collected bonuses on **PlayerTank** object. Accroding to this attribute behavior of the PlayerTank will be changed. PlayerTank can be faster, frozen bullet shooter, long range shooter or protected by a shield for the duration of the time.

private boolean hasShieldProtection: Attribute shows whether **PlayerTank** object has a **protective shield** or not which disable any damage on PlayerTank for a certain time interval.

Methods:

public void setShieldProtection(boolean hasShieldProtection): Funtion changes the condition of the **hasShieldProtection** attribute of PlayerTank object.

public boolean hasShieldProtection(): Function returns a boolean value which shows the state of **hasShiledProtection** attribute of PlayerTank. According to value that is returned from this function, GameEngine behaves differently when a collision between a Bullet and PlayerTank.

public void decrementMyBonusDuration(): Function decreases the **my_bonus_duration** attribute of the PlayerTank to control the effect of the Bonus which is collected. This function is used to shorten the effect of Bonus in terms of time.

public void incrementMyBonusDuration(): Function increases the **my_bonus_duration** attribute of the PlayerTank to control and makes the effect of the Bonus longer than normal time interval.

public void incrementLifeBonusDuration(): Function increases the **my_bonus_duration** attribute of the PlayerTank to control and makes the effect of the Bonus longer than normal time interval.

public long getMyBonusDuration(): Funtion returns the time value as a long variable which shows the remaining duration of **my_bonus_duration** attribute.

public void shootMetal(GameEngine engine): Function takes the **GameEngine** as input variable and creates a **MetalBullet** and add the new **Bullet** object to the **allObjectsList** of the GameEngine. GameEngine draws the Bullet on the screen and controls the movement of the Bullet.

public void shootIce(GameEngine engine): Function takes the **GameEngine** as input variable and creates a **IceBullet** and add the new **Bullet** object to the **allObjectsList** of the GameEngine. GameEngine draws the Bullet on the screen and controls the movement of the Bullet.

private void setIcons(): Function initializes each ArrayList of Image **life_x** attributes by using 20 different images which shows the health bar and the **PlayerTank** in four different directions and five different **healthBar** states.

public ArrayList<Image> getCurrentPlayerIcon(): Function controls the **life** attribute of **GameObject** in this context PlayerTank and selects the current **ArrayList** which shows the appropriate healtBar in four different direction. Function also returs the selected **ArrayList** for **GameEngine** to draw the PlayerTank.

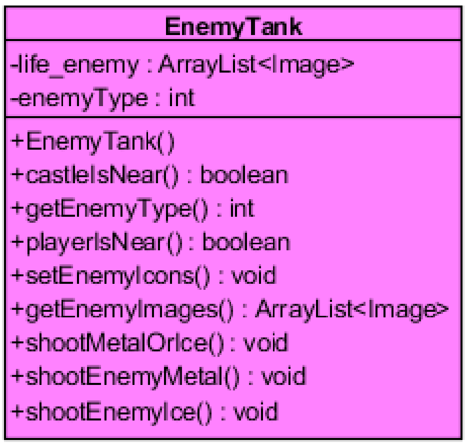
public ArrayList<Image> get5LifeIconImages() : Function returns **life_x** attributes as ArrayList of Image to draw the **PlayerTank** on the screen by **GameEngine** class. Other 4 implementation of these functions are below.

> **public ArrayList<Image> get4LifeIconImages() :**
>
> **public ArrayList<Image> get3LifeIconImages() :**

public ArrayList<Image> get2LifeIconImages() :

public ArrayList<Image> get1LifeIconImages() :

3.3.4 EnemyTank Class



EnemyTank

-life_enemy : ArrayList<Image>
-enemyType : int

+EnemyTank()
+castleIsNear() : boolean
+getEnemyType() : int
+playerIsNear() : boolean
+setEnemyIcons() : void
+getEnemyImages() : ArrayList<Image>
+shootMetalOrIce() : void
+shootEnemyMetal() : void
+shootEnemyIce() : void

An **EnemyTank** is an extension of the **Tank** class, so this class inherits all of the basic functionalities of the **Tank** class. It is a tank that moves independent of the user meaning it will be moving according to a programmed logic which aims at killing the users PlayerTank and its Castle. It can move around the map, shoot bullets but it cannot collect Bonus. There are several different **types** with each one harder and harder to eliminate with more speed and health as well.

Cosntructor:

public EnemyTank(boolean _isAlive, float _coordinateX, float _coordinateY, int width, int height, int life, int speed_of_tank, int enemyType):

This constructor initiates a new Enemy Tank that has a certain position and dimensions on the map and life, speed and enemyType which controls the speed, life and image of the Tank.

Attributes:

private ArrayList<Image> life_enemy: life_enemy attribute hold the images of any EnemyTank objects in four different directions. According to **enemyType**, images will be different and **life_enemy** attribute is initialized differently.

private int enemyType: this attribute specifies the type of the enemy tank. It actually represents which picture should be chosen to build the enemy tank image. So there will be several picture groups each representing a different type of an enemy corresponding to integers ranging from 1-4 for 4 types of enemy tanks respectively each with a different color.

Methods:

public boolean castleIsNear(): this method will calculate the distance between the EnemyTank and the users's castle. In case the enemy tank is currently inside a certain closed perimeter, the EnemyTank then will be notified with this method returning true. This will further enable the EnemyTank to shoot towards the users castle only if it is near.

public int getEnemyType() : Function returns the **enemyType** attribute of the **EnemyTank** object to determine which type of enemy is mentioned.

public boolean playerIsNear() : this method will calculate the distance between the EnemyTank and the PlayerTank. In case the EnemyTank is currently inside a certain closed perimeter around the PlayerTank, then EnemyTank will be notified with this method returning true. This will further enable the EnemyTank to shoot towards the PlayerTank only if it is near.

private void setEnemyIcons(int enemyType): Function takes the **enemyType** as an input to add appropriate images in four different directions to **life_enemy** array list of images. This array list is used to draw the EnemyTank object by **GameEngine** class during gameplay.

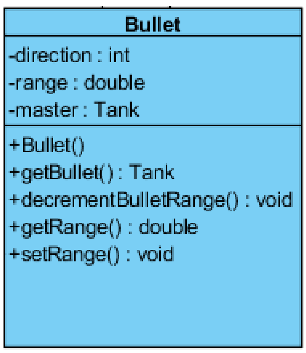
public ArrayList<Image> getEnemyImages() : returns the **life_enemy** attribute which hold the ArrayList of Images of certain type of enemy in four different direcctions. Returned ArrayList is used in **GameEngine**.

public void shootMetalOrIce(GameEngine engine): Function controls the **iceBullet** attribute of the **tank class** to control whether the Tank can shoot ice bullet or not .Accroding to this check function calls **shootEnemyMetal** or **shootEnemyIce** functions to create a **bullet**.

public void shootEnemyMetal(GameEngine engine): Function takes the **GameEngine** as input variable and creates a **MetalBullet** and add the new **Bullet** object to the **allObjectsList** of the GameEngine. GameEngine draws the Bullet on the screen and controls the movement of the Bullet.

public void shootEnemyIce(GameEngine engine): Function takes the **GameEngine** as input variable and creates a **IceBullet** and add the new **Bullet** object to the **allObjectsList** of the GameEngine. GameEngine draws the Bullet on the screen and controls the movement of the Bullet.

3.3.5 Bullet Class



This class extends the GameObject class with attributes and operations aimed to implement the functionalities of a bullet. Every **Tank** has its own **Bullet** with varying speed.

A bullet object has a destination **X** and **Y** which represent the point to which the bullet should travel. So after shot, the bullet than will disappear at destination point if it has not crashed with another GameObject.

Constructor

public Bullet(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height, Tank master, int direction, int range, int speed)

This constructor initiates a **Bullet** object which will have an initial position noted according to the X and Y coordinate system of the map as well as a destination position. Since we have different types of Bullets they must also have a certain **range and speed** value .Also the **Bullet** object should belong to its corresponding tank which is why a **Tank** object should be passed as a parameter in this constructor.

Attributes:

private int direction : the bullet can go in four directions, this attribute will be used in order to show which direction this bullet is going to.

private double range : this attribute is to be used when calculating the positions of the tank this bullet shooted by.

private Tank master: this Tank object represents the tank to which the bullet belongs. Since all the tanks are shooting bullets one must specify to which **Tank** each bullet belongs to, so we pass this object as a parameter in the **Bullet** class constructor.

Methods:

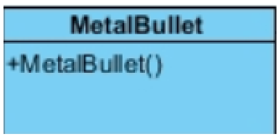
public Tank getBullet(): returns a **Tank** object, owner of the referenced **Bullet.** As mentioned and explained in the attributes above, the **Bullet** should have an owner (the one who shoots it). Since we will need to define the bullet owner in the **catchCollision** method in **GameObject** class we will be needing this method in advance.

public void decrementBulletRange(): there is a range that the bullet can go. To handle the transition of the bullet, this method updates the positions of the bullet.

public double getRange(): this method returns the range of the bullet. This can be used in the above methods therefore this method is useful.

public double setRange(): this method sets the range. This can be used in the above methods therefore this method is useful.

3.3.6 MetalBullet Class



The **MetalBullet** extends the **Bullet** class meaning it adds attributes to the parent class. It gives the object a special graphical look and also customized animations.

Constructor:

public MetalBullet (boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height, Tank master, int direction, int range, int speed)

This constructor creates a **MetalBullet** by calling the parent constructor from **Bullet** over the actual given parameters defining position, destination, damage and owner of the **Bullet.** Moreover an image is accessed locally and assigned to this object so that the **MetalBullet** can be easily distinguished from other type bullets. Furthermore this type of the bullet will cause the other **GameObjects** to lose one life point.

3.3.7 Ice Bullet Class

IceBullet
+IceBullet()

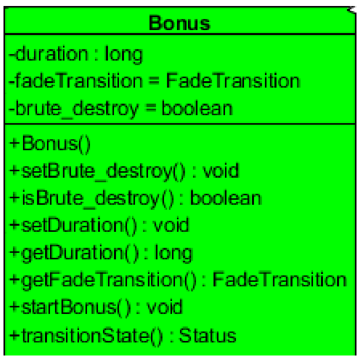
The **IceBullet** class extends the **Bullet** class as well in that it implements another type of bullet. In this case the bullet will have a white ball small icon and customized animations as well.

Methods:

public IceBullet (boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height, Tank master, int direction, int range, int speed)

This constructor creates a **MetalBullet** by calling the parent constructor from **Bullet** over the actual given parameters defining position, destination, damage and owner of the **Bullet.** Moreover an image is accessed locally and assigned to this object so that the **MetalBullet** can be easily distinguished from other type bullets. Furthermore this type of the bullet will disable the movements of the player tank.

3.3.8 Bonus Class


Bonus
-duration : long
-fadeTransition = FadeTransition
-brute_destroy = boolean
+Bonus()
+setBrute_destroy() : void
+isBrute_destroy() : boolean
+setDuration() : void
+getDuration() : long
+getFadeTransition() : FadeTransition
+startBonus() : void
+transitionState() : Status

This class must implement the Bonus object. The bonuses should appear in the map at a random location and they are disappeared after a fixed amount of time. If a **PlayerTank** object collides with a **Bonus** object, it will gain upgrades. The upgrades mean added speed, special bullets and extra protection from enemies. These tank upgrades however will expire after a certain amount of time. The **EnemyTank** cannot collect a Bonus thus nothing happens when an **EnemyTank** collides with a **Bonus** object.

Constructor:

public Bonus (boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height)

This constructor initiates a Bonus object with specific location parameters. The object has a duration which specifies how long the object will be available on the map for. In order to make this calculations we need to know the time at which the bonus was first appeared as well. The Bonus object lifespan ends when its duration expires. The Bonus object will be based on pre-designed images of a fixed size in pixels which will be accessed locally.

Attributes:

private long duration : this attribute stores the duration of the Bonus object on the map in milliseconds. When duration expires the Bonus' lifespan expires.

private FadeTransition fadetransition: this attribute will be used in getFadeTransition() method to make a bonus to fade since they are not available throughout the game constantly.

private boolean brute_Destroy: this attribute tells if the **bonus** is destroyed or not.

Methods:

public void setBrute_destroy(): with this method the bonus will disappear in the Map.

public boolean isBrute_destroy(): it returns whether the Bonus was destroyed or not.

public void setDuration(long duration): since the bonuses in the map does not show up forever, this method sets the time duration or in other words sets the lifespan of the bonus.

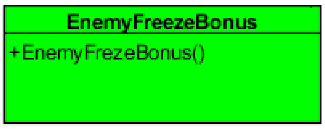
public long getDuration(): the bonuses have lifespans and this method returns the lifespan of a bonus.

public FadeTransition getFadeTransition(): since the bonuses in the map does not show up forever, this fadetransition method is useful by making the bonus to fade. JavaFx 's functionalities will be used in this class.

public void startBonus():bonuses start to show up in the map in some durations. When this method is invoked, it makes the bonus appear in the map.

public Animation.Status transitionState(): this returns an animation status whether the transition is done.

3.3.9 EnemyFreezeBonus Class



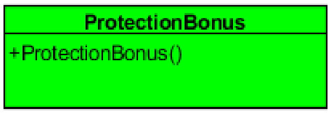
This class extends the Bonus class. It attains all of the attributes and the functionalities of a **Bonus** object but it differs in its special characteristics in that it specifies a custom Bonus upgrade option. When the **PlayerTank** object collides with an **EnemyFreezeBonus** it gains the ability to shoot several **IceBullet** (a Tank hit by an IceBullet cannot move for some seconds)

Constructor:

public EnemyFreezeBonus(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height)

This constructor initiates an **EnemyFreezeBonus** object with specific location parameters. The object has a duration which specifies how long the object will be available on the map for. The parameters are overridden to the parent class **Bonus** (super (…)). This constructor will also set a specific image (accessed locally) which corresponds to the **SpeedBonus** functionality such that the user can distinguish between Bonuses.

3.3.10 ProtectionBonus Class



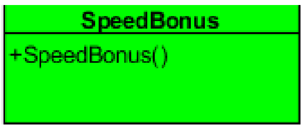
This class extends the Bonus class. It attains all of the attributes and the functionalities of a **Bonus** object but it differs in its special characteristics in that it specifies a custom Bonus upgrade option. When the PlayerTank collides with a **ProtectionBonus** its life points are not affected by any incoming enemy bullet at all. This type of Bonus has a limited lifespan as it expires after some seconds as well.

Constructor:

public ProtectionBonus(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height)

This constructor initiates a **ProtectionBonus** object with specific location parameters. The object has a duration which specifies how long the object will be available on the map for. The parameters are overridden to the parent class **Bonus**. This constructor will also set a specific image (accessed locally) which corresponds to the **ProtectionBonus** functionality such that the user can distinguish between Bonuses.

3.3.11 SpeedBonus Class



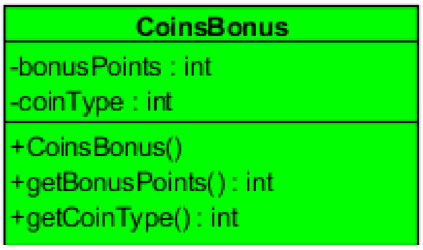
This class extends the Bonus class. It attains all of the attributes and the functionalities of a **Bonus** object but it differs in its special characteristics in that it specifies a custom Bonus upgrade option. When the **PlayerTank** object collides with a **SpeedBonus** its motion speed is increased and thus the tank can move a lot faster than normally. The effect of this upgrade disappears after some seconds as well

Constructor:

public SpeedBonus(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height)

This constructor initiates a **SpeedBonus** object with specific location parameters. The object has a duration which specifies how long the object will be available on the map for. The parameters are overridden to the parent class **Bonus** (super(…)). This constructor will also set a specific image (accessed locally) which corresponds to the **SpeedBonus** functionality such that the user can distinguish between Bonuses.

3.3.12 CoinsBonus Class


CoinsBonus
-bonusPoints : int
-coinType : int
+CoinsBonus()
+getBonusPoints() : int
+getCoinType() : int

With creating this class, we aimed to have a **Coins** object. As it can be understood from the name, this classes has Coins object and when the player reaches to that object, the player gains points. There are several coin types such as you can gain 100 coins in one catch or you can get 800 coins if you are lucky. To identify the type of coins, we specified an attribute called **coinType**.

Attributes:

private int bonusPoints: with this attribute, we can calculate how many point the player can get.

private int coinType: this attribute has a number which identifies the coin type such as '800'.

Constructor:

public CoinsBonus(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height, int coinType) :

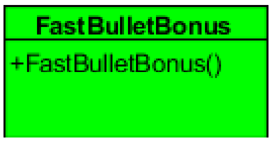
This constructor initiates a **CoinsBonus** object with specific location parameters.. The parameters are overridden to the parent class **Bonus** (super(…)). This constructor will also set a specific image (accessed locally) which corresponds to the **CoinsBonus** functionality such that the user can distinguish between Bonuses.

Methods:

public int getBonusPoints (): this method returns the points that the player can get when catching a coin.

public int getCoinType (): this method returns the type of the coin.

3.3.13 FastBulletBonus Class



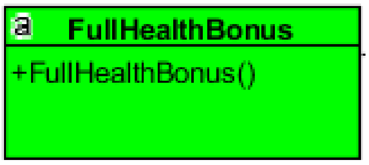
With this class, there will be a **FastBullet** bonus object. Like the bricks, we have four types of bonuses and this is one of them. In this type, when the bonus appears on the screen, and if the player can catch it, as it can be derived from the name, player can shoot fast bullets for a specific of a lifetime .

Constructor:

public FastBulletBonus(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height)

Since the fastbullet property, does not show up forever, we put the parameter _isAlive as the span of this bonus. Like the other game entities, with this constructor we create a FastBullet object by initializing its coordinates in the map.

3.3.14 FullHealthBonus Class



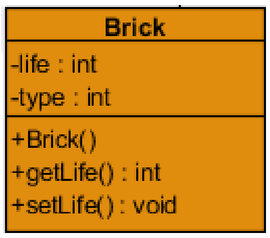
With this class, there will be a **FullHealth** bonus object. Like the bricks, we have four types of bonuses and this is one of them. In this type, when the bonus appears on the screen, and if the player can catch it, as it can be derived from the name, players health points increases. This bonus will be helpful if an enemy tries to kill the player.

Constructor:

public FullHealthBonus(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height)

Since the full health property does not show up forever, we put the parameter _isAlive as the span of this bonus. Like the other game entities, with this constructor we create a FullHealth bonus object by initializing its coordinates in the map.

3.3.15 Brick Class



This class is aimed at implementing a **Brick** object. Basically this object is supposed to become an obstacle to the tanks moving around the map. Yet, they can be destroyed if shot by a **MetalBullet**. The **Brick** has a position on the map which means the **Brick** has a certain (X, Y) coordinate on the game map.

Attributes:

private int life: each brick will have life points depending on the type of the brick. No life points means the brick state is dead thus the brick gets destroyed and deleted from the map.

private int type: since we have got four types of bricks, we need to define which type of brick we are using with this attribute.

Constructor:

public Brick (int life, int type):

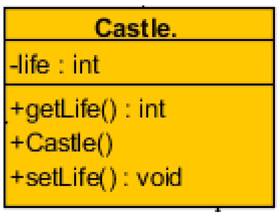
This constructor initiates a **Brick** object with specific location parameters. There are several types of bricks such as green brick, red brick, yellow brick and white brick. Each brick has a different characteristics, in order to have separate classes it is easier to have one for the implementation.

Methods:

public int getLife (): this method returns the life points of the brick. It will be used frequently on collision events.

public int setLife (): this method set the life points of the brick. It will be used frequently on collision events.

3.3.16 Castle Class



A Castle is an object which is stationary throughout all the gameplay. The castle is therefore related to the lifespan of the game. If the castle gets hit by bullets its life points decrease and when they reach 0 the castle is destroyed and the game/level ends. There will be two types of castles; **EnemyCastle** and **PlayerCastle.**

Constructors:

public Castle(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height,int life)

This constructor initiates a Castle object which will be situated at a certain position on a corner of the game map. The objects lifespan depends on the number of shots it gets hit and its **life** points.

Attributes:

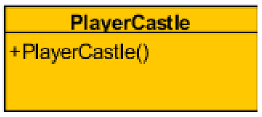
private int life: specifies the number of life points the corresponding castle has. Notice that for harder levels the Castle needs to have more life points than usual since the game would be very hard to play and win.

Methods:

private int getLife(): returns the number of **life** points that the castle currently has. Since this is the threshold to winning or losing the game, this method will be used frequently when bullet collisions are detected.

private int setLife(): since each castle has different life points, this method sets the life points of the castles when invoked.

3.3.17 PlayerCastle Class



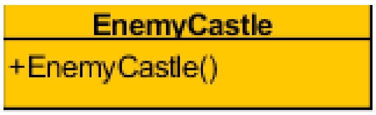
The **PlayerCastle** class builds up on a **Castle** object which belongs to the **PlayerTank.** This means that the **PlayerTank** must protect the castle from getting hit by the enemies since if the **PlayerCastle** is destroyed than **PlayerTank** loses the game.

Constructors:

public PlayerCastle(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height, int life)

This constructor initiates sets the default position(on the left of the map) and life points of a castle as well as assigning an image icon to the **PlayerCastle** object so that it is distinguished from the opponent’s castle.

3.3.18 EnemyCastle Class



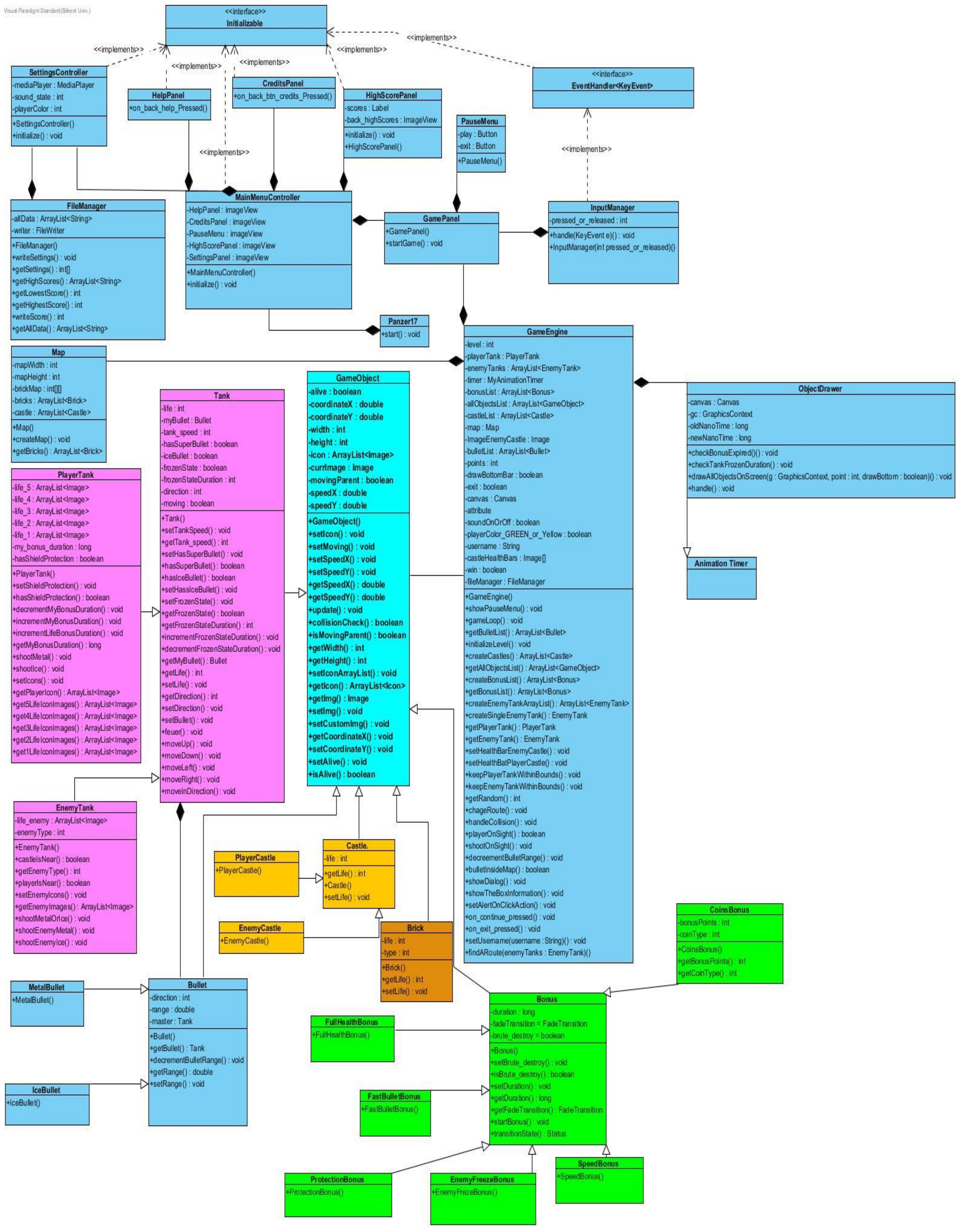
The **EnemyCastle** class builds up on a **Castle** object which belongs to the **EnemyTank.** This means that the **EnemyTank** must protect the castle from getting hit by the enemies (**PlayerTank**) since if the **EnemyCastle** is destroyed than **PlayerTank** wins the game.

Constructors:

public EnemyCastle(boolean _isAlive, double _coordinateX, double _coordinateY, int width, int height, int life)

This constructor sets the default position (on the right of the map) and life points of the castle as well as assigning an image icon to the **EnemyCastle** object so that it is distinguished from the opponent's castle. The **EnemyCastle** will have fixed life points which increases in every level making it harder for the game to be won.

4. Object Class Diagram


Visual Paradigm Standard(Bilkent Univ.)
<<interface>>
Initializable
<<implements>>
<<implements>>
<<implements>>
<<implements>>
<<implements>>
<<implements>>
<<interface>>
EventHandler<KeyEvent>
<<implements>>
SettingsController
-mediaPlayer : MediaPlayer
-sound_state : int
-playerColor : int
+SettingsController()
+initialize() : void
HelpPanel
+on_back_help_Pressed()
CreditsPanel
+on_back_btn_credits_Pressed()
HighScorePanel
-scores : Label
-back_highScores : ImageView
+initialize() : void
+HighScorePanel()
PauseMenu
-play : Button
-exit : Button
+PauseMenu()
FileManager
-allData : ArrayList<String>
-writer : FileWriter
+FileManager()
+writeSettings() : void
+getSettings() : int[]
+getHighScores() : ArrayList<String>
+getLowestScore() : int
+getHighestScore() : int
+writeScore() : int
+getAllData() : ArrayList<String>
MainMenuController
-HelpPanel : imageView
-CreditsPanel : imageView
-PauseMenu : imageView
-HighScorePanel : imageView
-SettingsPanel : imageView
+MainMenuController()
+initialize() : void
GamePanel
+GamePanel()
+startGame() : void
InputManager
-pressed_or_released : int
+handle(KeyEvent e)() : void
+InputManager(int pressed_or_released)()
Panzer17
+start() : void
Map
-mapWidth : int
-mapHeight : int
-brickMap : int[][]
-bricks : ArrayList<Brick>
-castle : ArrayList<Castle>
+Map()
+createMap() : void
+getBricks() : ArrayList<Brick>
GameEngine
-level : int
-playerTank : PlayerTank
-enemyTanks : ArrayList<EnemyTank>
-timer : MyAnimationTimer
-bonusList : ArrayList<Bonus>
-allObjectsList : ArrayList<GameObject>
-castleList : ArrayList<Castle>
-map : Map
-ImageEnemyCastle : Image
-bulletList : ArrayList<Bullet>
-points : int
-drawBottomBar : boolean
-exit : boolean
-canvas : Canvas
-attribute
-soundOnOrOff : boolean
-playerColor_GREEN_or_Yellow : boolean
-username : String
-castleHealthBars : Image[]
-win : boolean
-fileManager : FileManager
+GameEngine()
+showPauseMenu() : void
+gameLoop() : void
+getBulletList() : ArrayList<Bullet>
+initializeLevel() : void
+createCastles() : ArrayList<Castle>
+getAllObjectsList() : ArrayList<GameObject>
+createBonusList() : ArrayList<Bonus>
+getBonusList() : ArrayList<Bonus>
+createEnemyTankArrayList() : ArrayList<EnemyTank>
+createSingleEnemyTank() : EnemyTank
+getPlayerTank() : PlayerTank
+getEnemyTank() : EnemyTank
+setHealthBarEnemyCastle() : void
+setHealthBatPlayerCastle() : void
+keepPlayerTankWithinBounds() : void
+keepEnemyTankWithinBounds() : void
+getRandom() : int
+chageRoute() : void
+handleCollision() : void
+playerOnSight() : boolean
+shootOnSight() : void
+decreementBulletRange() : void
+bulletInsideMap() : boolean
+showDialog() : void
+showTheBoxInformation() : void
+setAlertOnClickAction() : void
+on_continue_pressed() : void
+on_exit_pressed() : void
+setUsername(username : String)() : void
+findARoute(enemyTanks : EnemyTank)()
ObjectDrawer
-canvas : Canvas
-gc : GraphicsContext
-oldNanoTime : long
-newNanoTime : long
+checkBonusExpired()() : void
+checkTankFrozenDuration() : void
+drawAllObjectsOnScreen(g : GraphicsContext, point : int, drawBottom : boolean)() : void
+handle() : void
Animation Timer
GameObject
-alive : boolean
-coordinateX : double
-coordinateY : double
-width : int
-height : int
-icon : ArrayList<Image>
-currImage : Image
-movingParent : boolean
-speedX : double
-speedY : double
+GameObject()
+setIcon() : void
+setMoving() : void
+setSpeedX() : void
+setSpeedY() : void
+getSpeedX() : double
+getSpeedY() : double
+update() : void
+collisionCheck() : boolean
+isMovingParent() : boolean
+getWidth() : int
+getHeight() : int
+setIconArrayList() : void
+getIcon() : ArrayList<Icon>
+getImg() : Image
+setImg() : void
+setCustomImg() : void
+getCoordinateX() : void
+setCoordinateY() : void
+setAlive() : void
+isAlive() : boolean
Tank
-life : int
-myBullet : Bullet
-tank_speed : int
-hasSuperBullet : boolean
-iceBullet : boolean
-frozenState : boolean
-frozenStateDuration : int
-direction : int
-moving : boolean
+Tank()
+setTankSpeed() : void
+getTank_speed() : int
+setHasSuperBullet() : void
+hasSuperBullet() : boolean
+hasIceBullet() : boolean
+setHasIceBullet() : void
+setFrozenState() : void
+getFrozenState() : boolean
+getFrozenStateDuration() : int
+incrementFrozenStateDuration() : void
+decrementFrozenStateDuration() : void
+getMyBullet() : Bullet
+getLife() : int
+setLife() : void
+getDirection() : int
+setDirection() : void
+setBullet() : void
+feuer() : void
+moveUp() : void
+moveDown() : void
+moveLeft() : void
+moveRight() : void
+moveInDirection() : void
PlayerTank
-life_5 : ArrayList<Image>
-life_4 : ArrayList<Image>
-life_3 : ArrayList<Image>
-life_2 : ArrayList<Image>
-life_1 : ArrayList<Image>
-my_bonus_duration : long
-hasShieldProtection : boolean
+PlayerTank()
+setShieldProtection() : void
+hasShieldProtection() : boolean
+decrementMyBonusDuration() : void
+incrementMyBonusDuration() : void
+incrementLifeBonusDuration() : void
+getMyBonusDuration() : long
+shootMetal() : void
+shootIce() : void
+setIcons() : void
+getPlayerIcon() : ArrayList<Image>
+get5LifeIconImages() : ArrayList<Image>
+get4LifeIconImages() : ArrayList<Image>
+get3LifeIconImages() : ArrayList<Image>
+get2LifeIconImages() : ArrayList<Image>
+get1LifeIconImages() : ArrayList<Image>
EnemyTank
-life_enemy : ArrayList<Image>
-enemyType : int
+EnemyTank()
+castleIsNear() : boolean
+getEnemyType() : int
+playerIsNear() : boolean
+setEnemyIcons() : void
+getEnemyImages() : ArrayList<Image>
+shootMetalOrIce() : void
+shootEnemyMetal() : void
+shootEnemyIce() : void
PlayerCastle
+PlayerCastle()
Castle.
-life : int
+getLife() : int
+Castle()
+setLife() : void
EnemyCastle
+EnemyCastle()
Brick
-life : int
-type : int
+Brick()
+getLife() : int
+setLife() : void
MetalBullet
+MetalBullet()
IceBullet
+IceBullet()
Bullet
-direction : int
-range : double
-master : Tank
+Bullet()
+getBullet() : Tank
+decrementBulletRange() : void
+getRange() : double
+setRange() : void
CoinsBonus
-bonusPoints : int
-coinType : int
+CoinsBonus()
+getBonusPoints() : int
+getCoinType() : int
Bonus
-duration : long
-fadeTransition = FadeTransition
-brute_destroy = boolean
+Bonus()
+setBrute_destroy() : void
+isBrute_destroy() : boolean
+setDuration() : void
+getDuration() : long
+getFadeTransition() : FadeTransition
+startBonus() : void
+transitionState() : Status
FullHealthBonus
+FullHealthBonus()
FastBulletBonus
+FastBulletBonus()
ProtectionBonus
+ProtectionBonus()
EnemyFreezeBonus
+EnemyFrezeBonus()
SpeedBonus
+SpeedBonus()

5.References

[1] https://docs.oracle.com/javase/8/javafx/api/toc.htm

[2] Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition*, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.*