



# CS 319 - Object-Oriented Software Engineering System Design Report

Panzer 2017

Group 2-C

Pınar BAYATA

Burak SİBİRLİOĞLU

Ndriçim RRAPİ

Ferhat Serdar ATALAY

# Contents

<b>1. Introduction</b>	4
1.1 Purpose Of The System	4
1.2 Design Goals	4
1.2.1 End User Criteria	4
1.2.2 Extensibility Criteria	4
1.2.3. Performance Criteria	5
1.2.4 Trade Offs	5
1.3 Overview	5
<b>2. Software Architecture</b>	6
2.1. Overview	6
2.2. Subsystem Decomposition	6
2.3. Architectural Styles	8
2.3.1 Layers & Model View Controller	8
2.4. Hardware / Software Requirements	9
2.5. Persistent Data Management	9
2.6. Access Control and Security	10
2.7. Boundary Conditions	10
<b>3. Subsystem Services</b>	11
3.1 User Interface Subsystem Services	11
3.1.1 Main Menu Class	12
3.1.2 GamePanel Class	13
3.1.3 PauseMenu	14
3.1.4 Credits Panel	14
3.1.5 Settings Panel	15
3.1.6 Help Panel	16
3.1.7 High Scores	17
3.2 Game Management Subsystem Interface	19
3.2.1 GameEngine Class	20
3.2.2 MapManager Class	23
3.2.3 InputManager Class	25
3.2.4 FileManager Class	26
3.3 Game Entities Subsystem Interface	28
3.3.1 GameObject Class	29

3.3.2 Tank Class.....	30
3.3.3 PlayerTank Class.....	32
3.3.4 EnemyTank Class.....	34
3.3.5 Bullet Class.....	37
3.3.6 GunBullet Class .....	38
3.3.7 Ice Bullet Class .....	39
3.3.8 Bonus Class .....	40
3.3.9 FastBullet Class .....	41
3.3.10 ProtectionBonus Class .....	42
3.3.11 SpeedBonus Class .....	43
3.3.12 EnemyFreezeBonus Class .....	44
3.3.13 Brick Class.....	45
3.3.14 GreenBrick Class.....	46
3.3.15 BlueBrick Class.....	47
3.3.16 BrownBrick Class.....	48
3.3.17 WhiteBrick Class.....	49
3.3.18 Castle Class .....	50
3.3.19 PlayerCastle Class .....	51
3.3.20 EnemyCastle Class .....	52
<b>4. Object Class Diagram .....</b>	<b>53</b>
<b>5.References .....</b>	<b>54</b>

# 1. Introduction

## 1.1 Purpose Of The System

Panzer 2017 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

Panzer 2017 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

Panzer 2017 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

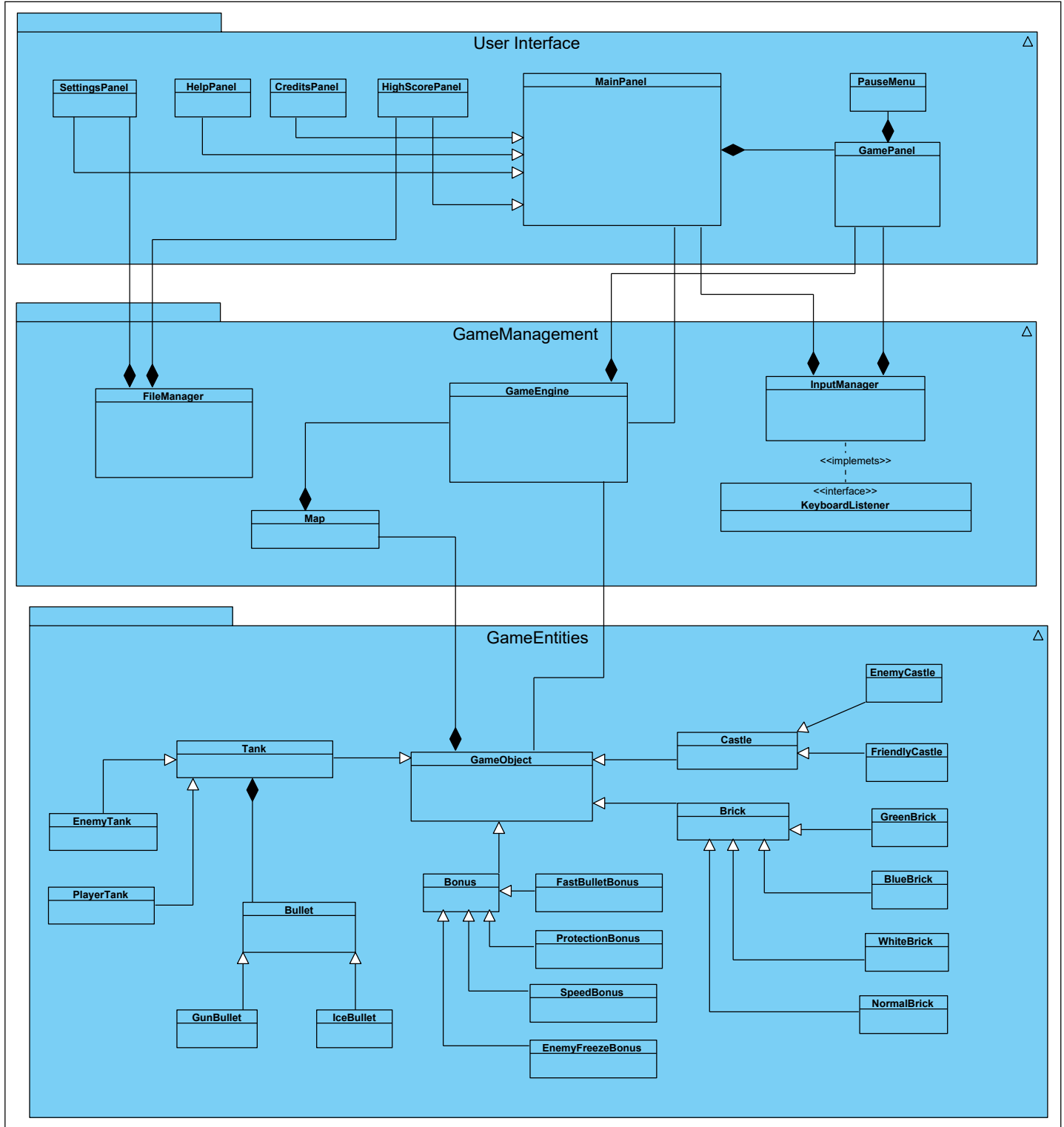
Maintaining and implementing a project needs manageable small subsystems.

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.



## 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.

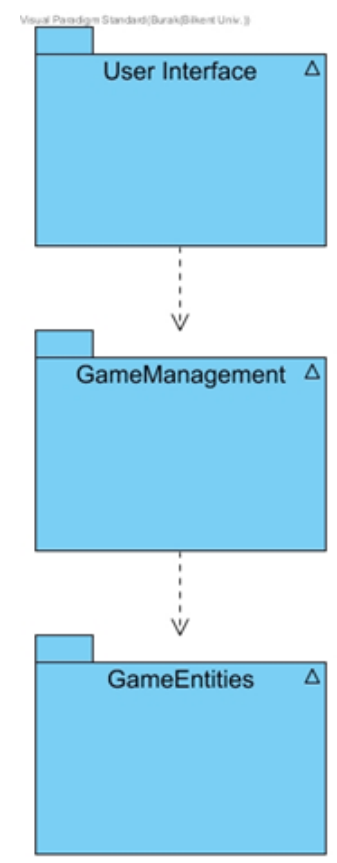


Figure 2: Decomposed System Layers



## 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 compiler 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. Instead 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.

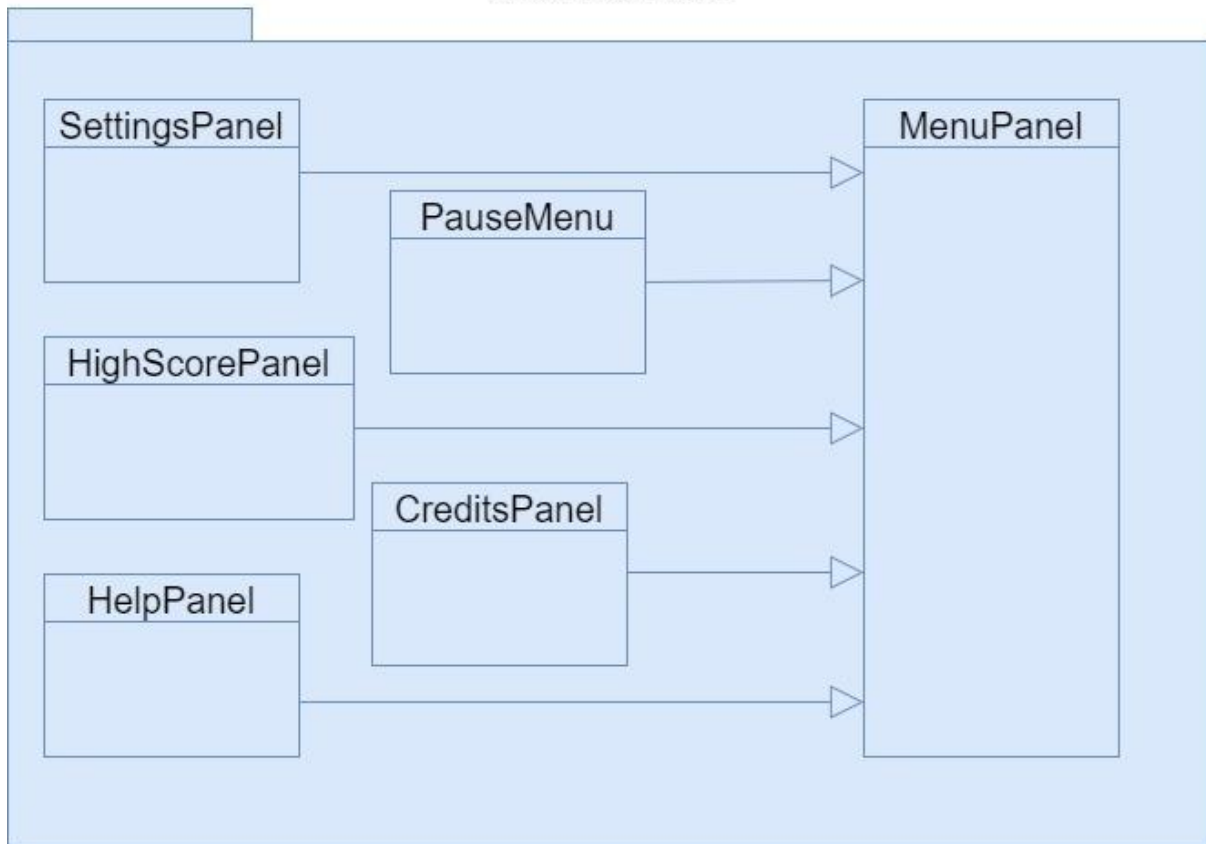
- **Error**

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

##### User Interface



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 Class

MainMenu
-playButton: Button -settingsButton: Button -helpButton: Button -scoreButton: Button -creditsButton: Button -backGround: Image -exitButton: Button

Main Menu 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.

#### *Constructor:*

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

#### *Attributes:*

**private playButton:** User has to click on the play button to start the game.

**private settingsButton:** Opens up the settings menu when clicked.

**private helpButton:** Opens the help screen.

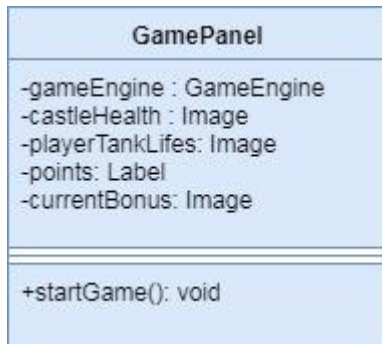
**private scoreButton:** Opens the highscore menu.

**private creditsButton:** Shows the credits screen.

**private backGround:** The background image for the main menu.

**private exitButton:** User can exit the game by clicking exit.

### 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.

#### *Constructor:*

**public GamePanel ():** The constructor for GamePanel. This constructor creates a panel which is going to hold **gameEngine**, **points**, and **Image** attributes.

#### *Attributes:*

**private GameEngine gameEngine:** The game panel triggers the Game Engine.

**private Image castleHealth:** Shows the health of the castles as an image.

**private Image playerTankLives:** Shows our remaining lives as the player in the picture format.

**private Label points:** Shows the points in the text format.

**private Image currentBonus:** Shows what type of bonus we have (if any) in picture format.

#### *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

PauseMenu
-playButton: Button -backGround: Image -exitButton: Button

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.

#### *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 playButton:** The button to return to gameplay

**private image backGround:** The background image for the pause menu.

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

### 3.1.4 Credits Panel

CreditsPanel
-credits: JText -backGround: Image -backButton: Button

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.

#### *Constructor:*

**public CreditsPanel ():** The constructor for Credits Panel. This constructor creates a panel which is going to hold **credits**, **backGround** and **backButton**.

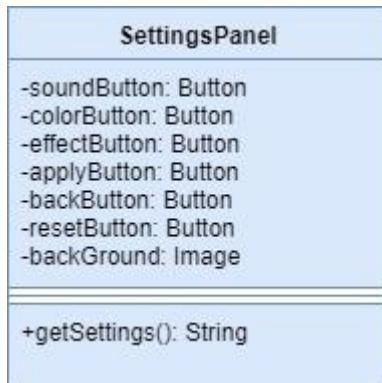
#### Attributes:

**private Label credits:** The text file holding the information of crew.

**private Image backGround:** The background image for the credits.

**private Button backButton:** The button for returning to main menu.

### 3.1.5 Settings Panel



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 enable or disable the sound of the game.
- Player can change the color of the tank.
- Player can disable shooting effects

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

#### 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 **backGround** and **Button** attributes.

#### Attributes:

**private Button soundButton:** Allows user to mute or unmute the sounds.

**private Button colorButton:** Allows user to change the color of the tank.

**private Button effectButton:** Allows user to change the effects.

**private Button applyButton:** After all options are decided, it must be clicked to apply the settings.

**private Button backButton:** The button for returning to main menu.

**private Button resetButton:** For returning to default settings.

**private Button backGround:** The background image for the settings menu.

#### *Methods:*

**public String getSettings():** This method will be using the **fileManager** to access the default settings which were previously saved in a text. This method returns a string of 2 characters, each of this characters represent the state of one setting. Ex: if received string is "10", 1 on the first digit place stands for sound is on, while 0 on the second digit place stands for color picked is yellow.

**public void setSettings(String data) :** Gets a string of data with the length 2 and containing only binary digits (for example; "10") with the first character belonging to the sound setting and the second char belonging to the **PlayerTank** color. Afterwards the changes are saved into the local text file through the **fileManager**.

### 3.1.6 Help Panel

HelpPanel
-instructions: JText -helpLogo: Image -backGround: Image -backButton: Button

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.



#### Constructor:

**public HelpPanel ():** The constructor for Help Panel. This constructor creates a panel which is going to hold **instructions**, **backButton** and **image** attributes..

#### Attributes:

**private Label instructions:** The instructions on how to play the game.

**private Image helpLogo:** The question mark image.

**private Image backGround:** The background image for the help menu.

**private Button backButton:** The button for returning to main menu.

### 3.1.7 High Scores

HighScorePanel
-first: JText -second: JText -third: JText -fourth: JText -fifth: JText-backGround: Image -backButton: Button -fileManager: FileManager
+getHighScore(): <ArrayList> String

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.

#### Constructor:

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

#### 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 Image backGround:** The background image for the highscores.

**private Button backButton:** The button for returning to main menu.

**private FileManager fileManager:** The File Manager object will be used to access and write into the text file which stores the high scores.

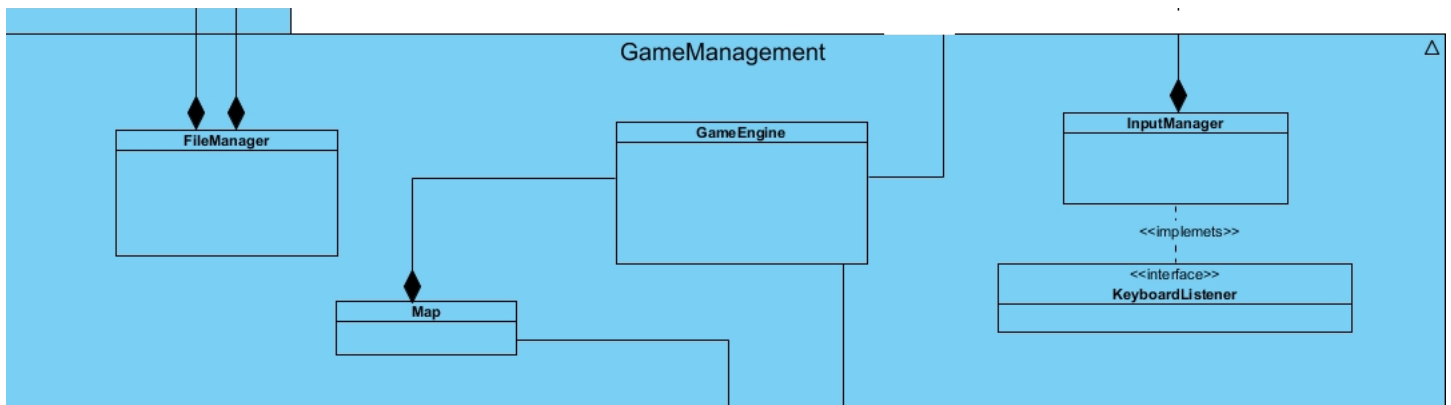
*Methods:*

**public String 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 a string of strings, including the names and the points of the player .

**private void setHighScores(String data) :** Saves new highscore into a text file through the **fileManager**.

## 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
<div><div>-level : int</div><div>-tankLevel : int</div><div>-Map : Map</div><div>-arrCastles : ArrayList</div><div>-arrEnemyTanks : ArrayList</div><div>-playerTank : PlayerTank</div><div>-arrBullets : ArrayList</div></div>
<div><div>+buildMap(level : int)</div><div>+checkCollision() : boolean</div><div>+produceCoin() : void</div><div>+destroyEnemy() : void</div><div>+destroyPlayer() : void</div><div>+upgradeTank() : boolean</div><div>+collectCoin() : int</div><div>+createEnemyTank() : void</div><div>+checkEnemyTanks() : int</div><div>+isGameCleared() : bool</div><div>+createPlayerTank() : boolean</div><div>+getPlayerCoordinates() : double</div><div>+gameLoop() : void</div></div>

This class is like the Façade class of the Game Management Subsystem. This class performs the proper operations according to the requests that came from User Interface subsystem, with the method gameLoop() this class runs the game in a loop.

#### *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 int tankLevel:** this attribute is used to know the player's tankLevel. Like the levels, tank level is cannot be selected by player so that the system will automatically increase the level when player upgrades a level.

**private Map map:** this attribute invokes Map class when needed. The map contains all of the positioning of the bricks in the screen and keeps track of deleted bricks as well.

**private ArrayList<castles> arrCastles:** this private attribute holds the castles objects in an array. This attribute is needed with the `gameLoop()` method when in the first run of the system.

**private ArrayList<tanks> arrEnemyTanks:** this private attribute holds the enemy tanks objects in an array. This attribute is needed with the `gameLoop()` method when in the first run of the system.

**private ArrayList<bullets> arrBullets:** : this private attribute holds the bullets objects in an array. This attribute is needed with the `gameLoop()` method when in the first run of the system.

**private PlayerTank playerTank:** as it can be seen from the type, this private attribute is needed to create a player tank object. In the first run of the system, player's tank place will be generated randomly.

#### *Constructors:*

**public GameEngine(int level, int tankLevel, Map map):** it initializes the attributes of the `GameEngine` for the first run of the system. For the first run, the level should be initialized as 1 as well as the tanklevel. Moreover, the map objects returns a map for level 1 for the first run of the system.

#### *Methods:*

**public void buildMap(level int):** this method interacts with `Map` class and loads the map according to the levels. Levels have different maps, so the system will update the map in each level automatically.

Also when a brick is collapsed, the system should update the map in every millisecond through a **gameLoop** which is managed by the **GameEngine**. So basically this method will be called inside the **gameLoop** that builds the screen's gameplay. So the loop method, which is provided by an instance of `Java` for dynamic games, is continuously building the map in each frame of a milliseconds so if a brick is destroyed at an instance the loop will

build the map again in the other millisecond frame thus the brick will be disappearing from the map.

**public boolean checkCollision():** this method returns true if a collision occurs. If the return value is 1, this method sends the message to the Map class and Map class will update the map in each collision.

**public void produceCoin():** this method creates coin objects and increases the number of the coins in the map. Coins are objects so this class interacts with Game Entities Subsystem.

**public void destroyEnemy():** this method removes the enemy's tank if it got shot. This method is also important for the Map class because the 'removing' action means changing the map after an enemy gets shot.

**public void destroyPlayer():** this method removes the player's tank if it got shot. As in the destroyEnemy class, this class is also important for the Map class because 'destroying' action means deleting the player's tank from the map. Also, after this method the game ends and Main Menu automatically shows up.

**public boolean upgradeTank():** when a level is updated, this method is called. This method upgrades tanks except the player's. This upgrade can happen after completing the level. That's why the constructor takes level as a parameter.

**public int collectCoin():** this method returns the number of coins collected by interacting with Map class. Coin object always remain in maps so the map is responsible for deleting coin objects after player collects coin.

**public void createEnemyTank():** this method creates an enemy tank object by interacting with Map class. As stated above, Map class is responsible for changing the maps to create enemy tank objects.

**public int checkEnemyTanks():** this method invokes the methods of this class which are designed to provide enemy tanks and finds the enemy tanks by one invocation to Map class.

**public boolean isGameCleared():**this method is invoked when a level is jumped. With invocation, Map class changes the map.

**public void createPlayerTank():**this method creates an enemy tank object by interacting with Map class. This method resembles to the createEnemyTank() method in such a way that the Map is responsible for changing the maps in the game.

**public double getPlayerCoordinates():** this function is called throughout the program to get the player's place in the game. This function always invoked because the game is played according to the player's place.

**public void gameLoop():** this method runs a loop in which the system is updated continuously. It is provided by Java's AnimationTimer [1] and lets us override the default method. So all of the game's visual will be handled in here, meaning as **checkCollision** returns true this method will make sure it deletes or adds the corresponding **GameObject** from the screen of gameplay.

**public void updateSettings():** Apart from holding the settings of the game, this method updates the settings of the game such as sound on or off, the color of the tank and enable or disable shooting effects.

### 3.2.2 MapManager Class

Map
-width : int
-enCastle : Castle
-brickArray : int[][]
-height : int
-plCastle : Castle
+drawMap() : void
+createMap()

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 width:** 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 Castle enCastle:** as it can be seen from the name of the attribute, this attribute references to the Castle object. enCastle means the enemy castle. Because we have two castles in the game, we put different object names to the castles. This castle will be seen in the Map.

**private int[][] brickArray:** 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. So, since the **gameLoop()** will be accessing the map at every second, when a spot on the 2D array goes from 1 to 0, the **gameLoop()** will simply stop drawing that brick in the next frame of millisecond.

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

**private Castle plCastle:** as it can be seen from the name of the attribute, this attribute references to the Castle object. plCastle means the player castle. Because we have two castles in the game, we put different object names to the castles. This castle will be seen in the Map.

#### *Constructors:*

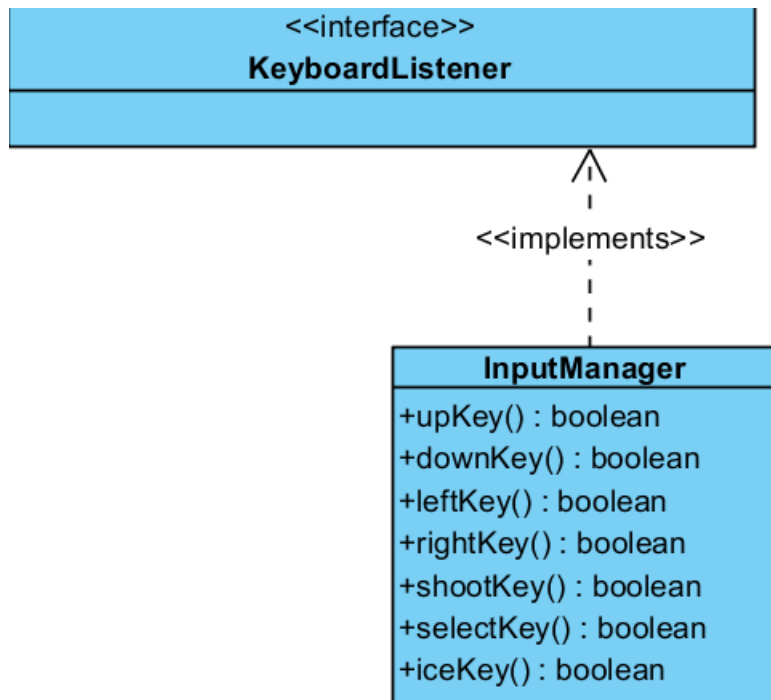
**public MapManager(int width, int height, Castle enCastle, Castle plCastle, int [] brickArray):** initializes a MapManager object with default attribute values.

#### *Methods:*

**public void drawMap(int width, int height):** draws the current map according to the given attributes.



### 3.2.3 InputManager Class



This class is designed to detect the user actions performed by keyboard(to move tanks on the screen, to pause the game, etc). In this context, this class implements proper interfaces of Java. This class is controlled by GameEngine class as well as the Map because in any of the results change, it will affect the other class.

#### *Constructors:*

**public InputManager():** it initializes the attributes of the InputManager for the first run of the system.

#### *Methods:*

**public boolean upKey():** this method returns true if the up key is pressed on the keyboard.

**public boolean downKey():** this method returns true if the down key is pressed on the keyboard.

**public boolean leftKey():** this method returns true if the left key is pressed on the keyboard.

**public boolean rightKey():** this method returns true if the right key is pressed on the keyboard.

**public boolean shootKey():** this method returns true if the space key is pressed on the keyboard.

**public boolean selectKey():** this method returns true if the enter key is pressed on the keyboard.

**public boolean iceKey():** this method returns true if the 'B' key is pressed on the keyboard. Icekey button can only be used after player gets an upgrade by jumping a level. Player with this facility, can shoot an enemy and freeze it.

### 3.2.4 FileManager Class

FileManager
-scoreList : ArrayList<score> -Level1Array : int [][] -Level2Array : int [][] -Level3Array : int [][]
+setNewEntry(user : string, score : int) : void +getList() : ArrayList<string> +resetList() : void

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<score> [] scores:** this array holds the highest score in an array.

**private int Level1Array[][]:** a 2d array that holds the position of each brick in the map for level 1. Each integer represents different type of brick such as green brick.

**private int Level2Array[][]:** a 2d array that holds the position of each brick in the map for level 2. Each integer represents different type of brick such as green brick.

**private int Level3Array[][]:** a 2d array that holds the position of each brick in the map for level 3. Each integer represents different type of brick such as green brick.

#### *Constructors:*

**public FileManager(ArrayList<score>[] scores, int Level1Array[][], int Level2Array[][], int Level3Array[][]):** 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 setNewEntry(user: string, score : int) :**this method adds a new user and user's score to the userList and the scoreList.

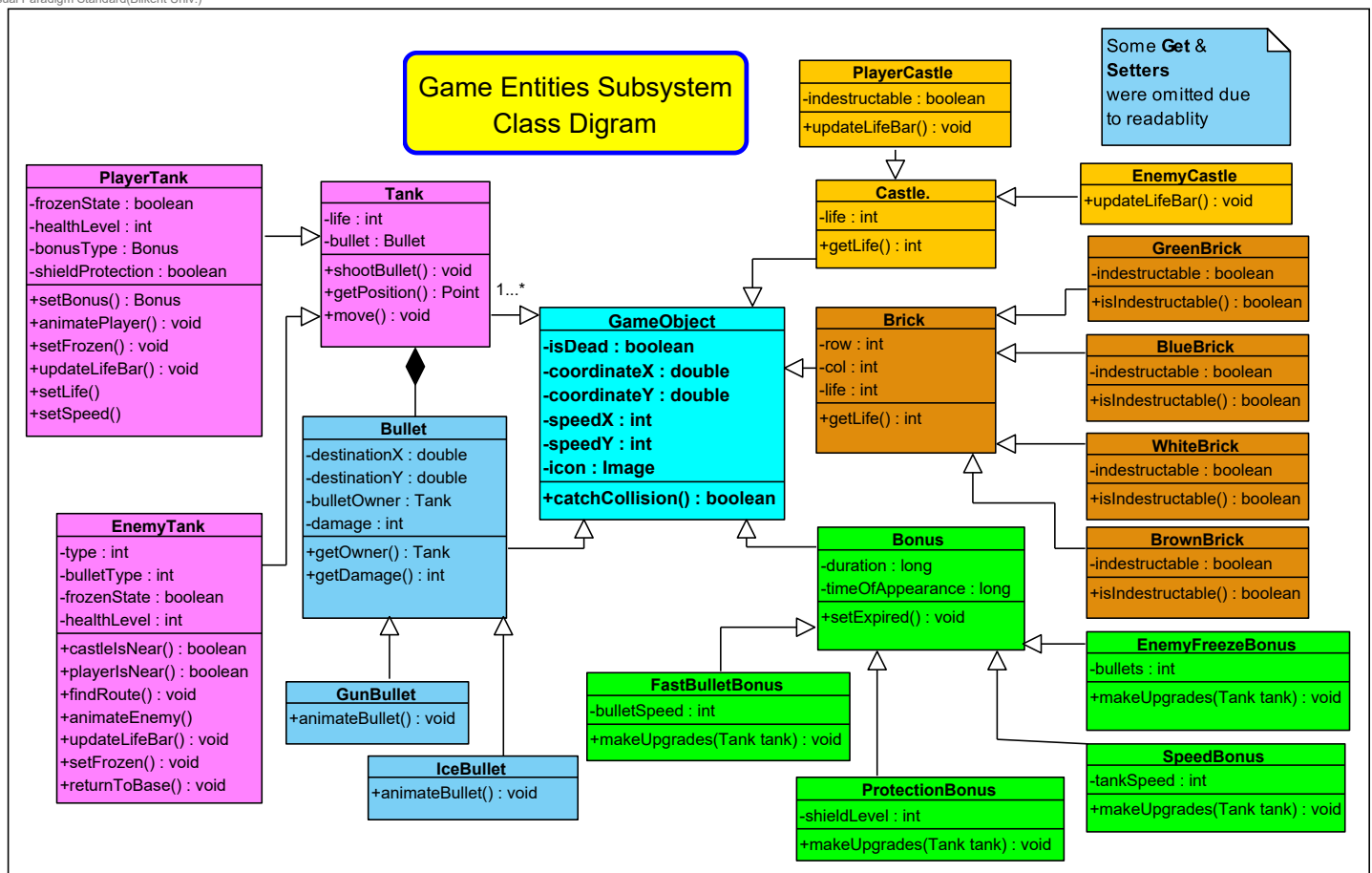
**public ArrayList<String> getList():** this method is invoked while interacting with the InputManager and returns the lists. This function gets the list from a text file and returns an array list of strings. The first 5 elements of the arraylist will correspond to the 5 highest scoring users and the 6 the element will contain the string data related to the settings of the game.

**public void deleteHighScoreList():** this method deletes the lists when invoked.

### 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 **Tanks**, **Bricks** and **Castles** onto the map. After the map is created the game starts and the user or enemies may initiate **Bullets** 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.

Visual Paradigm Standard(Bilkent Univ.)



### 3.3.1 GameObject Class

Visual Paradigm Standard (Bilkent Univ.)

GameObject
-isDead : boolean
-coordinateX : double
-coordinateY : double
-speedX : int
-speedY : int
-icon : Image
+catchCollision() : 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 (int posX, int posY, boolean isDead)**

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. All of the **GameObjects** can be destroyed with no exception.

#### *Attributes:*

**private boolean isDead:** this Boolean value specifies if the GameObject lives or not. If isDead is set to true 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)

**private int 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.

**private int 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 Image icon:** specifies the image 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.

#### *Methods:*

**private boolean catchCollision():** return true if a collision was detected; A collision is the overlapping of **GameObject**'s X and Y coordinates on the map. Since we do not want the Tank's to float over each other some action must be taken to stop that from happening. This method thus will make the necessary calculations for each collision scenario such as **Tank-Tank** collision, **Tank-Brick** collision, **Tank-Castle** collision or **Tank-Bullet** collision.

### 3.3.2 Tank Class

Visual Paradigm Standard (Bilkent Univ.)

<b>Tank</b>
-life : int -bullet : Bullet
+shootBullet() : void +getPosition() : Point +move() : void

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 points are not zero.

#### *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 bullet** : 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.

#### *Constructors:*

**public Tank(double posX, double posY, boolean isDead)**: this constructor initiates a tank at a certain specific location on the map (X,Y). When created a tank must have its position and life state "isDead" set to false.

#### *Methods:*

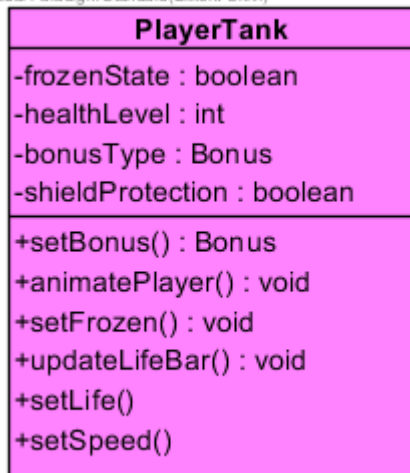
**public void shootBullet( )**: this method is called when the user presses the "Space" button on the keyboard. On the button press a Bullet object is created and moves toward the direction which the tank is pointed to. The bullet does not move infinitely but after it reaches a certain distance it gets self-destroyed, so that the players should approach each other instead of shooting from far away.

**public Point getPosition()** : this method returns the actual position of the tank in the map in Point-type format. In this way we can know both the X and Y coordinates of the tank if we need to access them in other methods.

**public void move()** : this method grants the tank the ability to move around in the map. It detects arrow button presses from the keyboard and moves the tank object accordingly up, down, left or right.

### 3.3.3 PlayerTank Class

Visual Paradigm Standard (Bilkent Univ.)



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, extra **health** but also **downgrades** such as **frozen** player state as well.

#### Constructor:

```
public PlayerTank(double posX, double Y, int life, int speed);
```

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

#### Attributes:

**private int healthLevel** : this attribute represents the health level of the user tank. The user will have a health bar at the top of his tank (a floating image representing current health). As he gets hit the health bar decreases in size indicating that the user has been shot and is losing health.

**private boolean frozenState**: this attribute specifies if the player tank was shot by an IceBullet or not. Some special enemy tanks can shoot Ice Bullets and if the user gets hit by those he will not be able to move or shoot for a specific duration.



**private ArrayList<Bonus> bonuses:** defines the bonuses the users tank has collected. A **Bonus** is an upgrade the user has collected throughout the game and as the game keeps going the user might be fast enough to collect several bonuses in the same time. Thus these bonuses need to be put in an array list and saved there until the bonus expires.

**private boolean shieldProtection:** this Boolean value gets true when the user has reached to a **Protection Bonus** object in the map and moved the tank over it (the bonus then disappears from the map). A shield means the user can get hit several times and not get injured. This occurs until the bonus time expires.

#### *Methods:*

**public void setBonus( Bonus bn):** this method will be used upon detection of collision in the **catchCollision** method of the main **FieldObject** class of this subsystem. It is thus called when **catchCollision** detects that the user's **PlayerTank** has collided with a **Bonus** object on the map and sets the **PlayerTank** newly acquired bonus accordingly.

**public void animatePlayer() :** this method will be used to indicate the direction of the tank. The player tank will have 4 direction states which will be represented by 4 different images. So when a user presses an arrow key from the keyboard the corresponding image must be summoned (ex: "left" key press changes the tank image to a tank image which points to the left and so on)

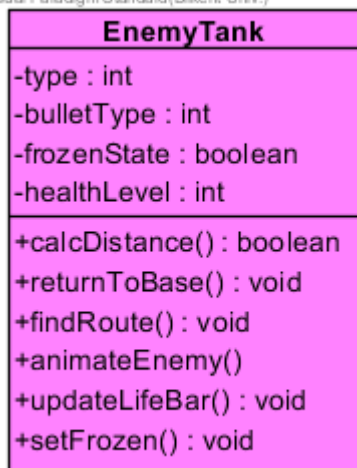
**public void setFrozen() :** this method will be called by the **catchCollision** method in case the user tank has had collision with an Ice Bullet. In such a scenario the user tank speed is set to 0 and thus the user may not move until the Ice Bullet loses its effect after some seconds. **public void updateLifeBar() :** this method will be used to indicate the user's tank health bar. As the user gets hit by bullets, its life minimizes. This is shown by a small health bar which is situated on top of each tank image. After each bullet hit this health bar shrinks and if the user gets killed the health bar would then totally disappear indicating that the user has lost all of the life's left.

**private void setLife(int num):** this method will be used to update the life count of the user tank. When the user gets a **Protection Bonus** his life points will increase by a certain constant. The user tank only retains this extra life points for a certain amount of time since when the bonus expires his life points are reset to the previous value.

**private void setSpeed(int num):** this method will be used to update the speed with which the user's tank can move around the map. It will be called when **catchCollision** method detects a collision between the user **tank** and a **Speed Bonus**. So this method will then increase the speed of the tank but the tank will get back to its normal speed when the **Speed Bonus** expires.

### 3.3.4 EnemyTank Class

Visual Paradigm Standard (Bilkent Univ.)



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(double posX, double Y, int life, int speed);**

This constructor initiates a new Enemy Tank that has a certain position on the map and full life points as well as a default constant speed which **never** changes.

#### *Attributes:*

**private int healthLevel** : this attribute represents the health level of the enemy tank. The enemy will have a health bar at the top of his tank (a floating image representing current health). As he gets hit the health bar decreases in size indicating that the user has been shot and is losing health. This health bar should differ in color with the users tank so that they can be easily distinguished from each other.

**private boolean frozenState**: this attribute specifies if the enemy tank was shot by an **IceBullet** or not. The user Player Tank can shoot Ice Bullets if it has previously collected an **EnemyFreezeBonus**. Thus when hit the enemy tank will not be able to move or shoot for a specific duration.

**private int type**: 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.

**private int bulletType**: this attribute specifies the type of the bullet this **EnemyTank** may shoot. Some of the enemy tanks will be able to shoot **IceBullets** instead of **GunBullets** so this integer type attribute will specify which bullet type this enemy tank will have. The integer choice selects between two different bullet images stored locally.

#### *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 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.

**public boolean returnToBase()** : this method will be used in upper levels to make the game even more difficult. It accesses the EnemyCastle **getLife** method to find out whether this castle is under attack or not. If the castle is getting hit then the method returns true and the enemy tank then has to return to his castle to protect it.

**public void animateEnemy()** : this method will be used to indicate the direction of the tank. The enemy tank will have 4 direction states which will be represented by 4 different images. So when the tank will be assigned to move in a certain direction the corresponding image must be summoned (ex: “a left pointing tank image” replaced with “a right pointing tank image”)

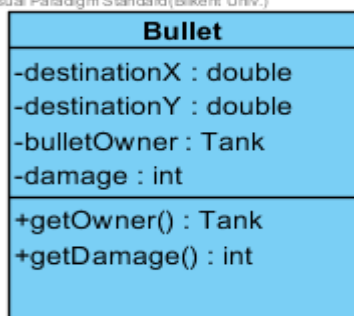
**public void updateLifeBar()** : this method will be used to indicate the EnemyTank’s tank health bar. As this tank gets hit by bullets, its life minimizes. This is shown by a small health bar which is situated on top of each tank image. After each bullet hit this health bar shrinks and if the user gets killed the health bar would then totally disappear indicating that the user has lost all of the life’s left.

**private void findRoute ()**: this method will be using the PlayerTank location to calculate which route the EnemyTank needs to take to get to its target. Notice that the path changes since the PlayerTank is always on the move so the method will be generating new paths. It will basically make the Enemy Tank move horizontally until the enemy and player have the same X coordinates. It will then move Vertically until the PlayerTank is close enough to be shot.

**private void setFrozen(Boolean state):** this method will be used to freeze the tank when it gets hit by an **IceBullet**. It will basically change the speed of the tank to 0 thus not allowing the tank to move at all. The method is called again when the effect of the **IceBullet** expires and the tank regains the ability to move normally.

### 3.3.5 Bullet Class

Visual Paradigm Standard (Bilkent Univ.)



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(double posX, double Y, double destX, double destY, int damage, Tank owner);
```

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. Having these attributes then we can animate the Bullet from point (**posX,posY**) to (**destX,destY**). Since we have different types of Bullets they must also have a certain **damage** value which is interpreted as the damage this type of bullet causes when hitting another **FieldObject**. 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 double destinationX** : marks the destination of the Bullet on the map's x-coordinate

**private double destinationY** : marks the destination of the Bullet on the map's y-coordinate

**private Tank bulletOwner**: 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.

**private int damage**: this attribute defines the severity level of the corresponding **Bullet**.

Since there will be different types of bullets each bullet will not cause the same damage when clashing to another **GameObject**. Therefore one bullet type may decrease the life of the opponent by 2 life points, 3 and so on.

#### *Methods:*

**public Tank getOwner()**: 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 int getDamage()** : returns the value of the **damage** attribute. Again this will be used in the **catchCollision** method in the **GameObject** class when a bullet hits a specific **GameObject**.

### 3.3.6 GunBullet Class

<b>GunBullet</b>
<b>+animateBullet() : void</b>

The **GunBullet** 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 GunBullet (double posX, double Y, double destX, double destY, int damage, Tank owner)**

This constructor creates a **GunBullet** 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 **GunBullet** can be easily distinguished from other type bullets. Furthermore this type of the bullet will cause the other **GameObjects** to lose one life point.

#### *Methods:*

**public void animateBullet():** specifies the particular animations for the corresponding bullet type. In this case when the GunBullet is shot specific small explosion animations will be put in place. These animations are composed of several pictures being rendered one after the other on separate time frames.

### 3.3.7 Ice Bullet Class

<b>IceBullet</b>
<b>+animateBullet() : void</b>

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 void animateBullet():** specifies the particular animations for the corresponding bullet type. In this case when the IceBullet is shot specific small white explosion animations will be put in place. These animations are composed of several pictures being rendered one after the other on separate time frames.

#### 3.3.8 Bonus Class

Bonus
-duration : long -timeOfAppearance : long
+getDuration() : long +setDuration() : void +getInitTime() : long +setInitTime() : void +setExpired() : void

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 (int posX, int posY, long duration, long timeOfAppearance)**

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 long timeOfAppearance** : this attribute stores the time at which the Bonus object will first appear on the map. The sum of the **duration** and **timeOfAppearance** defines the threshold at which the **Bonus** will disappear from the map.

#### *Methods:*

**public void setExpired():** this method sets the state of the Bonus to expired. The **Bonus** object inherits the **GameObject** attributes thus when we set the Bonus object to expired it means that the **isDead** (from **GameObject** class) attribute is set to true.

### 3.3.9 FastBullet Class

<b>FastBulletBonus</b>
-bulletSpeed : int
+makeUpgrades(Tank tank) : void

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 **FastBulletBonus** it can shoot bullets a lot faster than usual, until the bonus time expires.

#### *Constructor:*

**public FastBulletBonus (int posX, int posY, long duration, long timeOfAppearance, int bulletSpeed):**

This constructor initiates a **FastBulletBonus** 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 **FastBulletBonus** functionality such that the user can distinguish between Bonuses.

#### *Attributes:*

**private int bulletSpeed** : the bullet speed will represent the speed the bullets acquire after the PlayerTank has collected a FastBulletBonus

#### *Methods:*

**public void makeUpgrades(Tank tank):** this method applies the necessary upgrades to the PlayerTank that has just collided with the Bonus object on the map. In this case the upgrade increases the bullet shooting speed of the corresponding tank.

### 3.3.10 ProtectionBonus Class

ProtectionBonus
-shieldLevel : int
+makeUpgrades(Tank tank) : void

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 (int posX, int posY, long duration, long timeOfAppearance, int shieldLevel):**

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.

#### *Attributes:*

**private int shieldLevel** : the shield level represents the level of the shield. The first level (0) protects the PlayerTank from **GunBullets** but not from **IceBullets**. The **shieldLevel** should have a value of 1 so that the PlayerTank is immune to all types of **Bullets**

#### *Methods:*

**public void makeUpgrades(Tank tank):** this method applies the necessary upgrades to the PlayerTank that has just collided with the Bonus object on the map. In this case the upgrade makes the PlayerTank immune to the incoming enemy bullets. The **shieldLevel** is chosen at random so that the user may not know whether the shield protects him/her from **IceBullets** as well or not. This is aimed at making the game more interesting and challenging.

### 3.3.11 SpeedBonus Class

SpeedBonus
-tankSpeed : int
+makeUpgrades(Tank tank) : void

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 ProtectionBonus (int posX, int posY, long duration, long timeOfAppearance, int tankSpeed)**

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.

#### *Attributes:*

**private int tankSpeed** : the tank speed represents the speed which the PlayerTank will acquire after collecting this type of bonus. The value of the **tankSpeed** is going to be random but always greater than the default tank speed.

#### *Methods:*

**public void makeUpgrades(Tank tank):** this method applies the necessary upgrades to the PlayerTank that has just collided with the Bonus object on the map. In this case the upgrade increases the speed with which the PlayerTank moves around the map. The **tankSpeed** is chosen at random so that the user may not know how much extra speed his/her tank is going to acquire. This is aimed at making the game more interesting and challenging.

### 3.3.12 EnemyFreezeBonus Class

EnemyFreezeBonus
-bullets : int
+makeUpgrades(Tank tank) : void

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 (int posX, int posY, long duration, long timeOfAppearance, int bullets):**

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.

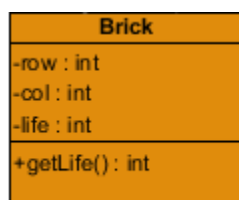
#### Attributes:

**private int bullets** : this attribute represents the number of **IceBullets** that the **PlayerTank** will acquire after collecting this type of bonus. Its value is going to be chosen randomly.

#### Methods:

**public void makeUpgrades(Tank tank):** this method applies the necessary upgrades to the **PlayerTank** that has just collided with the **Bonus** object on the map. In this case the upgrade enables the **PlayerTank** to shoot a number of **IceBullets**. The number of **bullets** is chosen at random so that the user may not know how much **IceBullets** his/her tank has got left.

### 3.3.13 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 **GunBullet**. The **Brick** has a position on the map which means the **Brick** has a certain (X, Y) coordinate on the game map.

### *Constructor:*

**public Brick (int posX, int posY, int row, int col):**

This constructor initiates a **Brick** object with specific location parameters. The first two parameters specify the location of the brick in the screen by pixels as inherited from **GameObject** while row and col define at which row and column of the map is the brick located at. Since the map will be divided into a grid of rectangles (will be using a 2D array). Each of these rectangles is designed such that it may fit a **GameObject**. Since we have a large number of **Bricks** we avoid naming each brick separately so we use row and col attributes instead. (Ex: we can access a brick at brickArray[2][5])

### *Attributes:*

**private int row:** this attribute defines at which row on the brick grid will the brick be situated.

If we want to delete a brick later we can simply access its specific row and column instead.

**private int col:** this attribute defines at which column on the brick grid will the brick be situated. If we want to delete a brick later we can simply access its specific row and column instead.

**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.

### *Methods:*

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

## 3.3.14 GreenBrick Class

GreenBrick
-indestructable : boolean
+isIndestructable() : boolean

This class inherits the attributes and methods from **Brick** class. **GreenBrick** should therefore be a brick with a green color and certain amount of life points distinguishable from the other types of bricks.

#### *Constructor:*

**public GreenBrick (int posX, int posY, int row, int col):**

This constructor initiates a **GreenBrick** object. So the object will not only have a position on the map as well a position on the brick grid but it will also have a preset image icon to itself as well. This image is accessed from local storage and it is the main graphical element that distinguishes this object from the other ones. In this case the image to be used is an icon of a green brick. Other than that the **GreenBrick** will have 1 life points meaning it can be destroyed only after 1 bullet shot.

#### *Attributes:*

**private boolean indestructable:** specifies whether the brick type is destroyable or not. In this case it is set to false, so this brick type can be destroyed

#### *Methods:*

**private boolean isIndestructable():** returns the boolean value of the variable **indestructible**. It is to be frequently used on collision checks.

### 3.3.15 BlueBrick Class

BlueBrick
-indestructable : boolean
+isIndestructable() : boolean

This class inherits the attributes and methods from **Brick** class. **BlueBrick** should therefore be a brick with a blue color and certain amount of life points distinguishable from the other types of bricks.

#### Constructor:

**public BlueBrick (int posX, int posY, int row, int col):**

This constructor initiates a **BlueBrick** object. So the object will not only have a position on the map as well a position on the brick grid but it will also have a preset image icon to itself as well. This image is accessed from local storage and it is the main graphical element that distinguishes this object from the other ones. In this case the image to be used is an icon of a **blue** brick. Other than that the **BlueBrick** will have 2 **life** points meaning it can be destroyed only after 2 bullet shots.

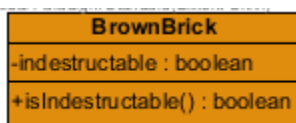
#### Attributes:

**private boolean indestructable:** specifies whether the brick type is destroyable or not. In this case it is set to false, so this brick type can be destroyed in two bullet shots.

#### Methods:

**private boolean isIndestructable():** returns the boolean value of the variable **indestructible**. It is to be frequently used on collision checks.

### 3.3.16 BrownBrick Class



BrownBrick
-indestructable : boolean
+isIndestructable() : boolean

This class inherits the attributes and methods from **Brick** class.

**BrownBrick** should therefore be a brick with a brown color and certain amount of life points distinguishable from the other

types of bricks

#### Constructors:

**public BrownBrick (int posX, int posY, int row, int col):**

This constructor initiates a **BrownBrick** object. So the object will not only have a position on the map as well a position on the brick grid but it will also have a preset image icon to itself as well. This image is accessed from local storage and it is the main graphical element that distinguishes this object from the other ones. In this case the image to be used is an icon of a



**brown** brick. Other than that the **BrownBrick** will have 3 life points meaning it can be destroyed only after 3 bullet shots.

#### *Attributes:*

**private boolean indestructable:** specifies whether the brick type is destroyable or not. In this case it is set to true, so this brick type cannot be destroyed regardless of the bullet shots it takes.

#### *Methods:*

**private boolean isIndestructable():** returns the boolean value of the variable **indestructible**. It is to be frequently used on collision checks.

### 3.3.17 WhiteBrick Class

WhiteBrick
-indestructable : boolean
+isIndestructable() : boolean

This class inherits the attributes and methods from **Brick** class.

**WhiteBrick** should therefore be a brick with a white color and

certain amount of life points distinguishable from the other types of bricks

#### *Constructors:*

**public WhiteBrick(int posX, int posY, int row, int col):**This constructor initiates a **WhiteBrick** object. So the object will not only have a position on the map as well a position on the brick grid but it will also have a preset image icon to itself as well. This image is accessed from local storage and it is the main graphical element that distinguishes this object from the other ones. In this case the image to be used is an icon of a **white** brick. Other than that the **WhiteBrick** will be indestructible and not affected by any bullet hits.

#### *Attributes:*

**private boolean indestructable:** specifies whether the brick type is destroyable or not. In this case it is set to true, so this brick type cannot be destroyed regardless of the bullet shots it takes.

#### Methods:

**private boolean isIndestructable():** returns the boolean value of the variable **indestructible**. It is to be frequently used on collision checks.

#### 3.3.18 Castle Class

Castle.
-life : int
+getLife() : int

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 (int posX, int posY, 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.

### 3.3.19 PlayerCastle Class

PlayerCastle
-indestructable : boolean
+updateLifeBar() : void

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 (int posX, int posY, 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.

#### *Attributes:*

**private boolean indestructable:** there will be times when the PlayerCastle will be immune to bullets, based on a special **Bonus** collected or other possible scenarios. This attribute will therefore determine whether the **PlayerCastle** can be destroyed or not.

#### *Methods:*

**Private void updateLifeBar():** this method will be used to update the state of the health bar corresponding to the **PlayerCastle**. Each time the castle loses life points it should be reflected on the health bar. The health bar is located on the bottom left corner of the screen alongside with other information such as points/bonuses collected.

### 3.3.20 EnemyCastle Class

EnemyCastle
+updateLifeBar() : void

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 (int posX, int posY, int life):**

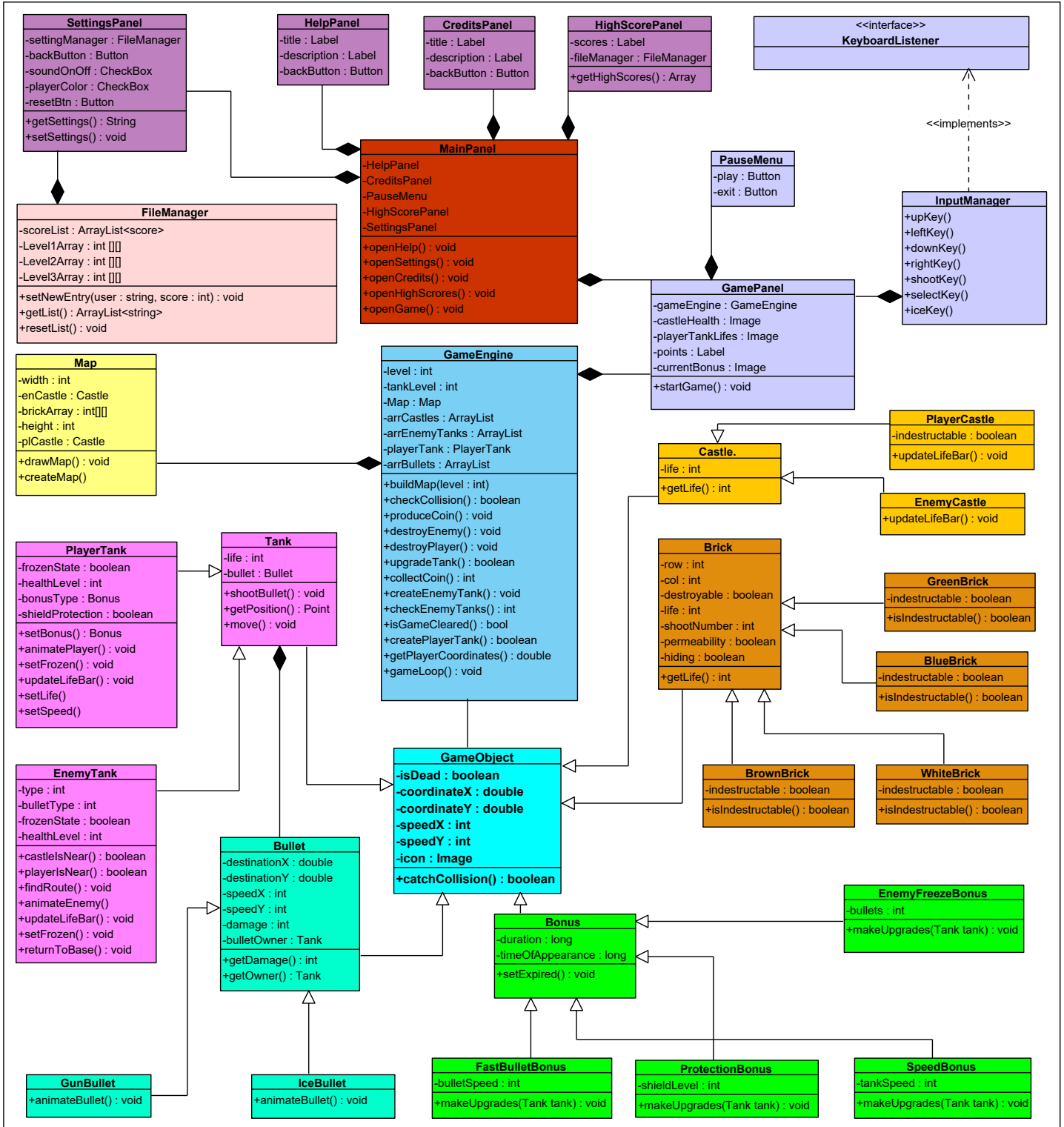
This constructor sets the default position (on the right of the map) and life points of the castle as well as assigning an image icon to the **EnemyCastle** object so that it is distinguished from the opponent's castle. The **EnemyCastle** will have fixed life points which increases in every level making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

#### *Methods:*

**private void updateLifeBar():** this method will be used to update the state of the health bar corresponding to the **EnemyCastle**. Each time this castle loses life points it should be reflected on the health bar. The health bar is located on the bottom right corner of the screen alongside with the game's other status information.

## 4. Full Object Class Diagram

Visual Paradigm Standard(Burak(Bilkent Univ.))



## 5. References

[1] <https://docs.oracle.com/javafx/2/api/javafx/animation/AnimationTimer.html>

[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.*