CS 319 - Object-Oriented Software Engineering

System Design Report

Battle City

Group 1-B

Kaan Sancak

Mahin Khankishizade

Ozan Kerem Devamlı

Supervisor: Ugur Dogrusoz

Table of Contents

1. Introduction

1.1 Purpose of the system

Battle City is a 2-D tank destruction and attack game. Its design is implemented in such ways that, the users could get the possible maximum satisfaction from the game. Its difference from the original game is, the level ends when all the other enemy tanks are destroyed. Battle City is planned to be a portable, user-friendly and challenging game, which aims to entertain the users by involving them into the survivor game. The player's goal is to destroy all the enemy tanks in each level in order to pass to the next level. It aims to be enough reflexive, fast, user-friendly and with a high performance engine.

1.2 Design goals

Design is another important step for creating the system. It helps to identify the design goals for the system that we should focus on. As it was mentioned in our analysis report, there are many non-functional requirements of the system that needs to be better clarified in the design part. In other words, they are the object of the focus in this paper. Following sections are the descriptions of the important design goals.

1.2.1 Trade-Offs

Development Time vs Performance:

For a game, most of people thinks that C++ presents the best performance, but also C++ is really hard to develop a game. Since C++ does not let to user control heap easily, we preferred to implement our project in JavaFX. This is reduced the development time since we did not bothered with memory leaks but also reduced the performance.

Space vs Speed:

Since our game needs to be used in real time, we preffered using much space to keep the speed high. We kept same variables at the different points of memory in order to decrease the search time for them and keeping CPU to be busy with calculating game physics and logic.

Functionality vs Usability:

Battle City is a game that presents basic fun of controlling a tank against bots. Since we are creating a game with not so complex interactions, we prefered usability over functionality. Our game is really simple to understand, and also has a how to play screen to develop usability far better.

Understandability vs Functionality:

As it was mentioned before, our system is expected to be easy to learn and understandable game. In respect to this, we had to decrease the complexity and functionality of the game by eliminating confusing and complicated options (i.e. too many options for power-ups, many options for keyboard input etc.) so that, the players would not bother to understand them, but would enjoy the simple understandable game.

Memory vs Maintainability:

During the design and the analysis of the game, we tried the factor out the common parts of the game objects and make use of abstraction as much as possible. Our main goal in this design thinking was to maintain the game objects easily as possible. However, by doing that we some object will have some methods and attributes which they do not need to have. This unnecessary attributes will cause memory allocation more than the game needs for to run a map level. Therefore, the abstraction we have done might cause unnecessary memory usage.

On the other hand, since we abstracted the common attributes as much as possible, it would be very easy to maintain the game which will result in better user experience. By benefiting these common attributes of subclasses, we hope that collisions, object updates will be handled easily.

Development Time vs User Experience:

During the analysis of the "Battle City" game, we have decided to implement interface components and graphics with JAVAFX libraries because JAVA's standard SWING library uses 8 bits for resolution on the other hand JAVAFX uses 32 bits for resolution. We have decided to use JAVAFX however it is harder for us to implement and develop the game with JAVAFX libraries because SWING is commonly used and known library which offers us more resources. By choosing implementation with JAVAFX, development time will be a lot harder for us on the other hand with benefits of JAVAFX, we will have better graphics and user experience.

1.2.2 Criteria

End User Criteria

Usability:

Battle City is expected to be an entertaining game for the users, which will provide them with a user-friendly interface, so that it was easy for the players to use it. For example, the simple and understandable menu and interface of the game will help the player to focus on the game itself, rather than understanding the game functions. The system will get the keyboard inputs from the users, which is the easy use for the players.

Ease of Learning: It is obvious that, most of the players skip the 'how to play' part of the game, almost every time. However, since our system is created in such a way that, it was easy for the players to understand it even without the help of how to play screen.

From the player's point of view, it will be very easy to identify and understand how the

level is ended, when the tank fires, attacks, how it moves, how the power-up life bonuses effect the tank etc.

Performance:

Performance is one of the important design goals, which is needed for the games in general. We are using GUI library for the better enhancement of the performance.

Maintenance Criteria

Extendibility:

The design of the game lets us to modify and change its features and functionalities in the future depending on the user feedback or other reasons. For example, we can extend its challenges by adding the time span for each level, or the strength differing between enemies and the player's tanks.

Modifiability:

Battle City game is designed as multilayered. With this design structure, it is easy to modify the system. Since the subsystems are connected weakly, the modification in one subsystem does not affect the other one. Thus, in our system, it is easy to modify the existing functionalities.

Reusability:

There are some subsystems (i.e. GUI) which are possible to be used in other games, without any change. For their reusability, they were designed independent from the system.

Portability:

Portability is an important point for a software in general, because it

enlarges the range of the game's usability for users. In other words, it helps the game to be played in various devices. Thus, we decided to implement in Java, since its JVM lets the platform to be independent and the system to be portable.

Performance Criteria

Response Time: It is important for the game to have the immediate reflex and reaction for the users' requests. Battle City game is designed so that its reactions were almost immediate, during the game itself, in display of the animations and effects.

1.3 Definitions

& Acronyms & Abbreviations

Java Virtual Machine (JVM) – an abstract computing machine for running a Java program.

Model View Controller (MVC) – the design pattern that we are going to use.

Graphic User Interface (GUI) – the library that we are going to code for the game.

2. System Architecture

2.1. Subsystem Decomposition

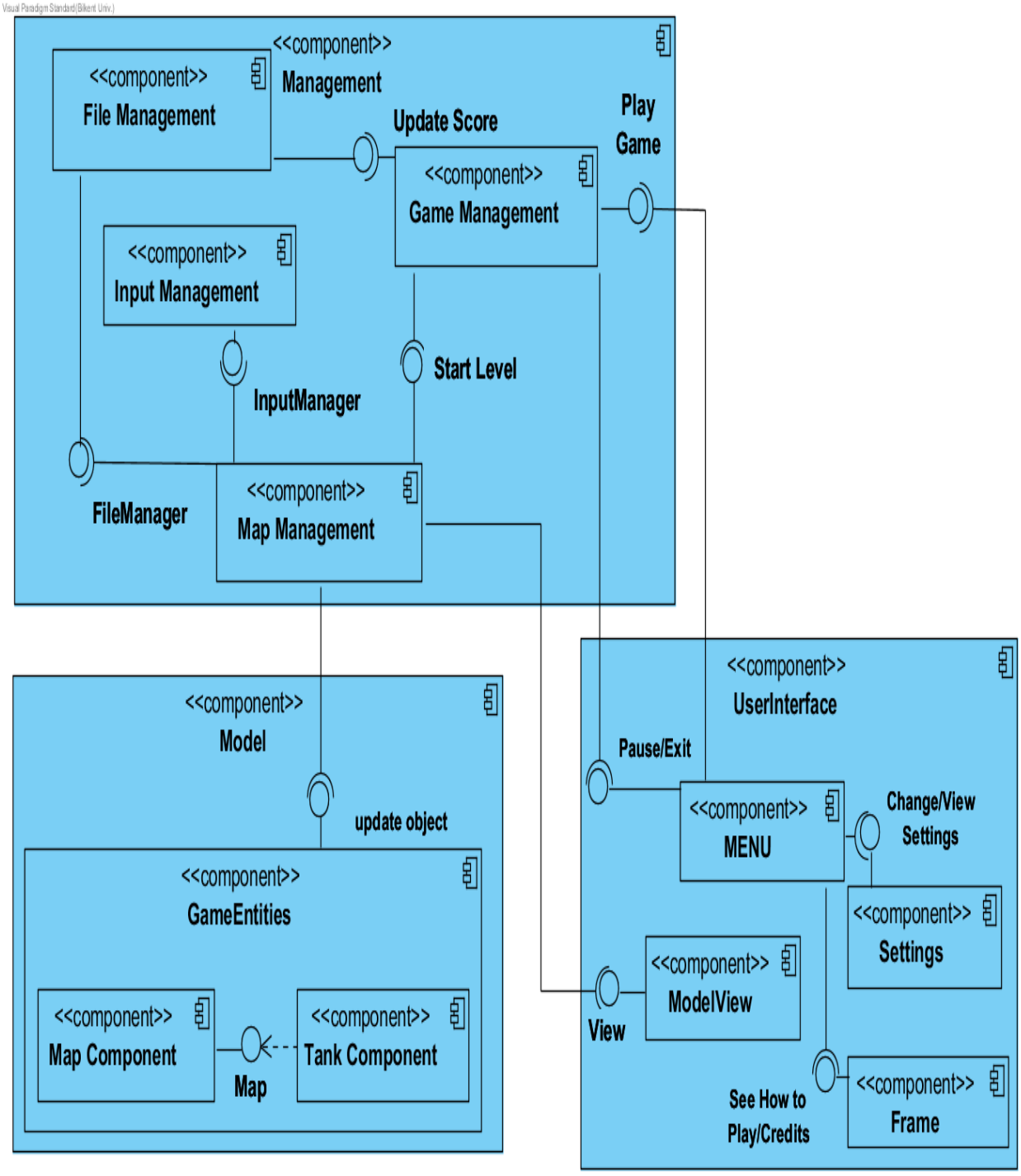


Figure 1-Subsystem Decomposition

In this section, we will decompose our system into subsystems. During this decomposition, our purpose is to reduce the coupling between different subsystems of the main system and increasing coherence of the components. By decomposing the game system as described we can easily modify the game or extend it when it is needed.

During the decomposition of the system, we have decided that **MVC(Model-View-Controller)** system design pattern is great fit to apply on our system. We divided our system based on MVC principles(See Figure-1 at top).

MVC is a suitable design pattern because of the following reasons:

- Our system exactly has three subsystems which we can match with MVC model.
    - We have **UserInterface Subsystem** which is our **View** includes some user interface components and provides user interface for the player.
    - We have our **Management Subsystem** which is our **Controller** and includes management components for the game. By using these components, **Management Subsystem** manages and handles the game. Management subsystem controls **game loop**.
    - We have our **Model Subsystem** which is our **Model** which includes **Game Entities Components.** Model subsystem includes every game object of the game and it is controlled and updated by **Management Subsystem**.
- Our dependencies and relationships between classes can be easily represented by MVC design pattern.
- MVC is a simple and effective design pattern which will increment the efficiency of the implementation stage.

We will go into more detail in continues parts.

As mentioned before, we divided our system into three subsystems which are represented in Figure-1. These three subsystems are User Interface subsystem, Management subsystem and Model subsystem. We tried to decompose these systems according to their different

functionalities. On the other hand, each subsystem has a responsibility to invoke other systems, so that game can stay maintainable during runtime. Meaning that,

- **UserInterface subsystem** can only request a functional action to Management subsystem by requesting 'play game' to Game Management component of Management Subsystem. Therefore any interaction between User Interface subsystem and Game Management subsystem must occur through GameManagement component. This makes very easy to solve any errors that may occur also it makes the game more stabilized, modifiable and extendable. It also holds the view component of the models. So that when the update comes from the management subsystem it updates the model views.
- **Management subsystem** acts like a controller of the game. Basically it has 4 components.
    - Two of them are responsible for handling the user input (**Input Management)** and saving/loading files(**File Management).**
    - The other two components which are **Game Management** and **Map Management** are more crucial for the **Management Subsystem.** Basically these two components handle the game. **Game management** component has functionalities for starting/ stopping/ ending the game. On the other hand, **Map Management** component responsible for controlling the **Model** and most importantly it controls the **Game Loop, Collisions** and **Model updates**.
    - We will go into more details in next chapters.
- **Model Subsystem,** includes components for game entities which are basically game objects. According to the data coming from controller they update their views in UserInterface subsystem.

As a result, our system decomposition will provide us high cohesion and low coupling. Having a system as described will make "Battle City" more flexible and extendable game

2.2. Hardware/Software Mapping

Battle City game will be implemented in Java programming language; more specifically we will use JAVAFX libraries. That's why, as software requirement "Battle City" will require Java Runtime Environment in order to be executed by other users and also "Battle City" will require at least the Java Development Kit 8 or a newer version because these versions include JAVAFX libraries.

Moreover, "Battle City" will require a keyboard as a hardware requirement. Keyboard will be used for I/O tool of the game. Users will control the menu selections, tank movements and also fire interactions with keyboard inputs. As a results of these two requirements, our game's system requirements will be minimal, just a computer with necessary software installed will be enough to compile the game and run it.

For storage of the game, meaning storage of the maps, highest scores, user setting preferences, we will use text files. Therefore, "Battle City" will not require any internet connection or database to operate.

2.3. Persistent Data Management

Battle City does not require any complex data storage system or database. Battle City will store the game instances, like objects, maps etc. in hard drive of the client. Meaning that, we will keep the game data in text files. Some of the text files like map designs will be instantiated during the implementation stage and these files cannot be modified. On the other hand, some data members like user setting preferences, sound settings, and highest score will be kept in a modifiable text files, therefore user can change or update the values of these files during game time. Moreover, to store the sound data we will use .waw format and to store the images we will use .gif format.

2.4. Access Control and Security

As mentioned at section 2.2 and 2.3, Battle City will not use any internet/network connection or any database. Basically, after loading the game and starting Battle City, anybody can play the game. Therefore, there will not be any access control and security measurement in order to prevent data leaks or any malicious actions. The highest scores

will be unique for each computer; meaning that highest scores will not kept the highest score made in the all of the computers but it will be kept unique for each computer.

2.5. Boundary Conditions

Battle City will not require any installation; it will not have an executable .exe extension. Battle City game will have an executable jar and the will be executed from .jar. This will also bring portability to the game. The game can be copied from a computer to another computer easy and in a status that ready to play.

Battle City game can be terminated by clicking “Exit” button in the main menu. In another scenario where player wants to quit during game time, user should pause the game by pressing “p” from keyboard or clicking “pause” button from game screen. After that, user can terminate the game by clicking “Exit” button in the main menu.

If there will be error during the file read, the game may start without sound and images. This problem can be solved by validating data members of the game and fixing the corrupted text file data.

If game collapses during game time because of an performance or design issue, the data(settings/score) will be lost.

3. Subsystem Services

3.1. UserInterface Subsystem

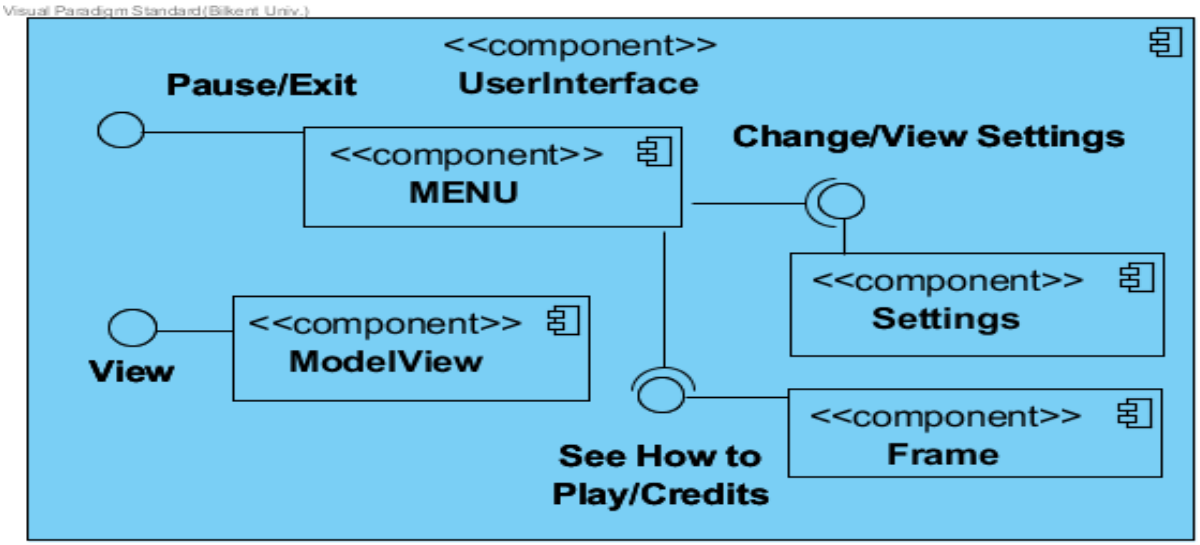


Figure 2- UserInterface Subsystem

UserInterface Subsytem is responsible for providing user interface for BattleCity game. It has three major components which are:

1. Menu Component
2. Frame Component
3. Settings Component
4. ModelView Component

When the user first starts the game, UserInterface provides a menu through its **Menu component.** Menu component has classes for different types of menus like Pause Menu and Stop Menu. Each of them has different responsibilities for UserInterface subsystem. Moreover, menu component can also invoke and instantiate other components like frame component and settings component. Here we can say that Menu Component is the interface which provides different functionalities for the interface.

ModelView Component, provides interface for model views. It gets updates from Management subsystem according to the changes in model subsystem.

Most importantly, only through **Menu Component** a new game (singleplayer/multiplayer) can be instantiated. In our system there are no other components that can instantiate a new game. Menu Component can be invoked by stop/exit game signals coming from **Management**

Component. Menu component achieves functionality of starting a new game through **Menu Class** which invokes **Game Management** subsystem by calling its methods which starts a new game. We can say that **Façade Design** is applied to reduce coupling and increment coherence.

Moreover, **Frame Component** provides some user interfaces for important information frames like How to Play, Credits and Confirmation Box. Frame component can only be invoked by Menu Component(**Façade Design Pattern**)

Lastly, **Settings Component** provides and instantiates settings for the user. Settings Component is responsible for keeping the user settings and changing them if there are any changes applied by the user. Again **Settings Component** can only be invoked by Menu component through **Main Menu Class(Façade Design Pattern).**

3.2. Management Subsystem

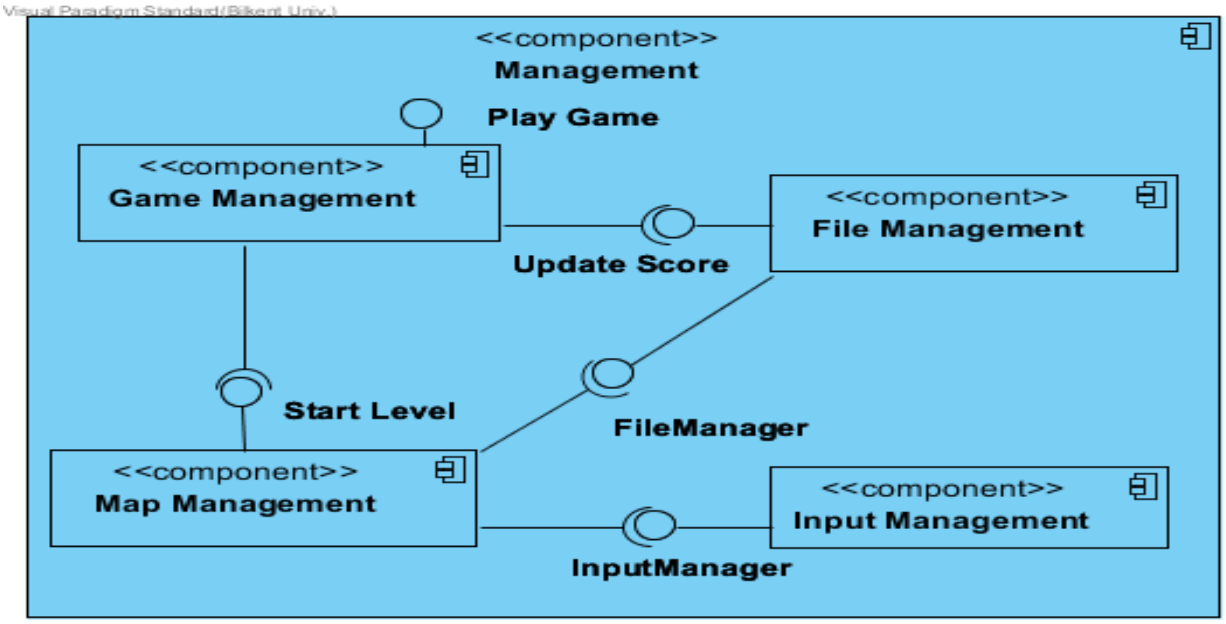


Figure 3- Management Subsystem

Management Subsystem is responsible for controlling and handling the game. It's component's controls the **Game Loop** of BattleCity which will be explained more detailed. Currently, it consists of 4 major components which are:

1. File Management
2. Game Management

3. Map Management
4. Input Management

Firstly, **FileManagement** subsystem is responsible for saving/loading files which can be in different formats (txt, mp3, jpg…). It is invoked by Map Management subsystem at beginning of game to get level objects and at end of game by Game Management to update highest score.

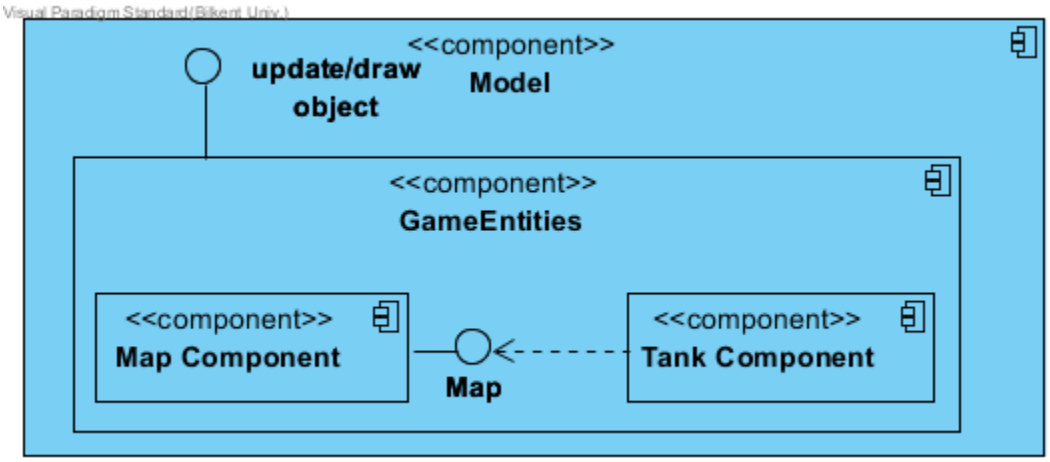
Game Management component is responsible for getting the play game input from **UserInterface** subsystem and instantiating a new game. To divide the workload of each component we have divided the management of game from management of map, that's why **Game Management** is only responsible for starting a new game, initializing next level when level is completed, invoking **Menu Component** when game is paused/exited, invoking File Management when new highest score achieved by updating score.

Map Management component is responsible for handling the Model and its components. Since we wanted a stabilized game as possible we have decided to use **Singleton Design Pattern** for **Map Management** component. Briefly, Map Management:

- It is the component which has **game loop**. Through Map Management component model and its components are constantly updated in a given time.
- It updates the **ModelView** component of **UserInterface** subsystem.
- It is the component which has **collision checks.** Through Map Management possible collision of the game frequently checked and if there is any possible collisions, Map Management updates/removes/damages the model objects accordingly through **Collision Manager** (Explained more detailed in low level design).
- Because of the attributes of Map Management given above, we decided to use **Singleton Design Pattern** in classes of **Map Management** because we though if we allow more than one map management object could cause handling problems.

Input Management handles the user input, if a user gives any specific input management notifies the map management and Map Management decides what to do next accordingly.

3.3. Model Subsystem



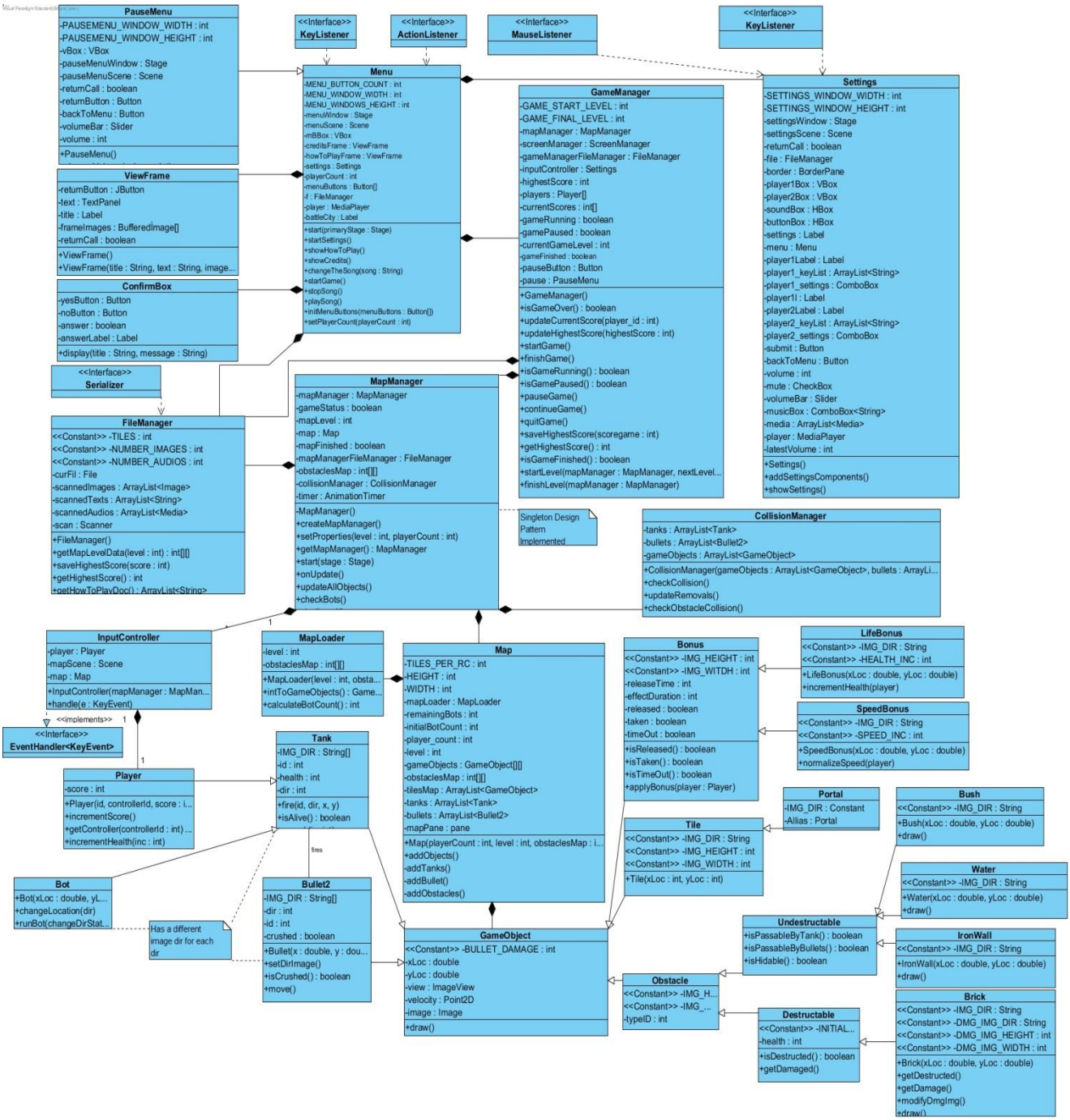
Model Subsystem represents our models. It consists of our Game Objects which can be different types. Model has a main component called **GameEntities** component which represent different types of game objects/entities. Basically all game entities can only be updated and drawn by the method call coming from **Management Subsystem**. Other than management component's Map Management component no other component or a class can modify the Model Component's instances. Model class has one component which includes two subcomponents:

- **Game Entities**: As mentioned before it represents the model objects.
    - **Map Component:** Represents the Map, its tiles and different obstacles and includes their operations.
    - **Tank Component:** Represents the Tanks which might be player's tank or a bot tank and includes its operations.
- According to the control flow coming from controller model updates it's view by sending calling proper methods which makes changes in View subsystem.

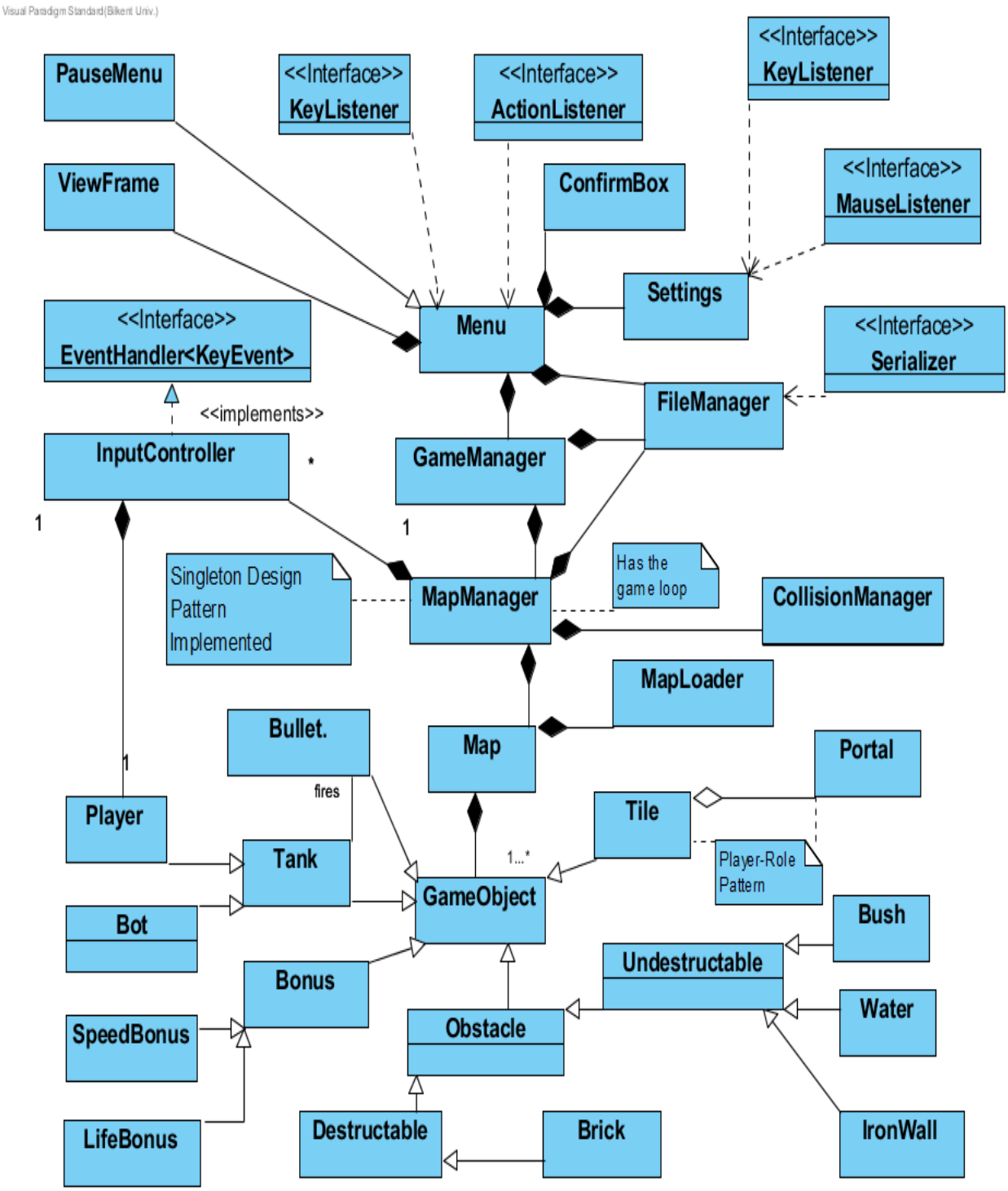
4. Low-level Design

4.1. Final object design

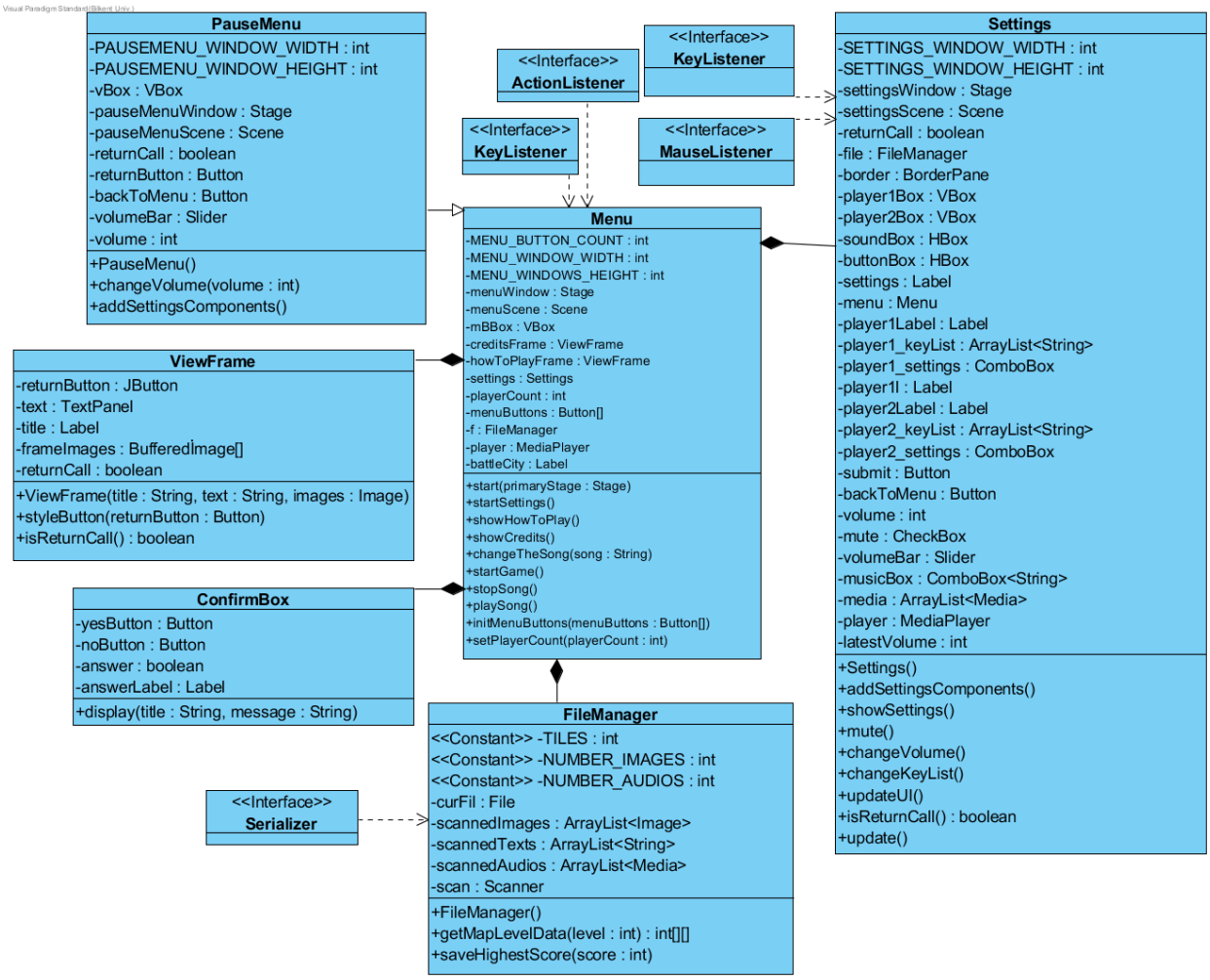
We will give the complete picture of final object design. After that we will give them in pieces to clear out the diagram since it is too big to fit in just one screen.



Abstraction of final object design(all class diagram collapsed):

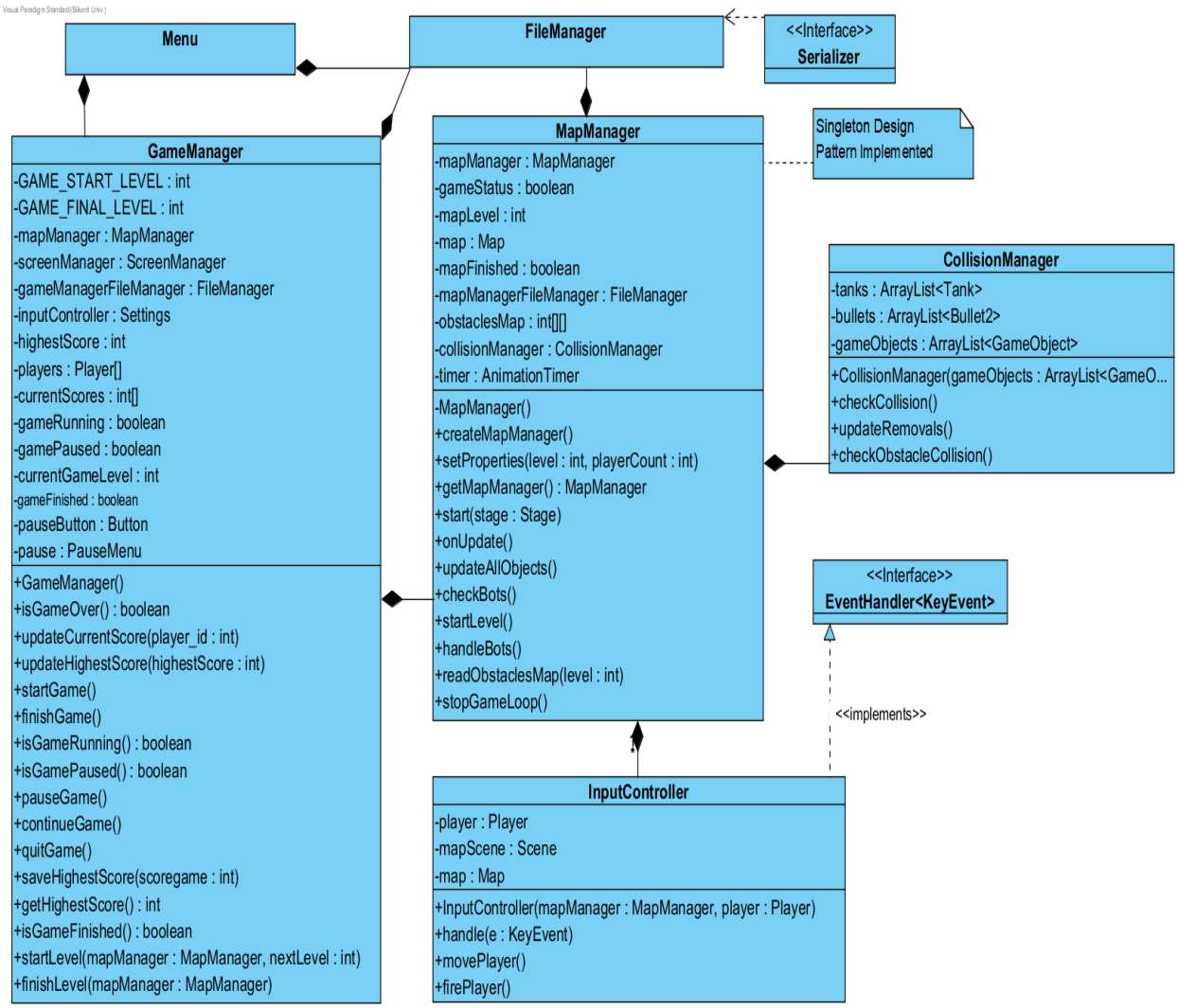


Piece 1:

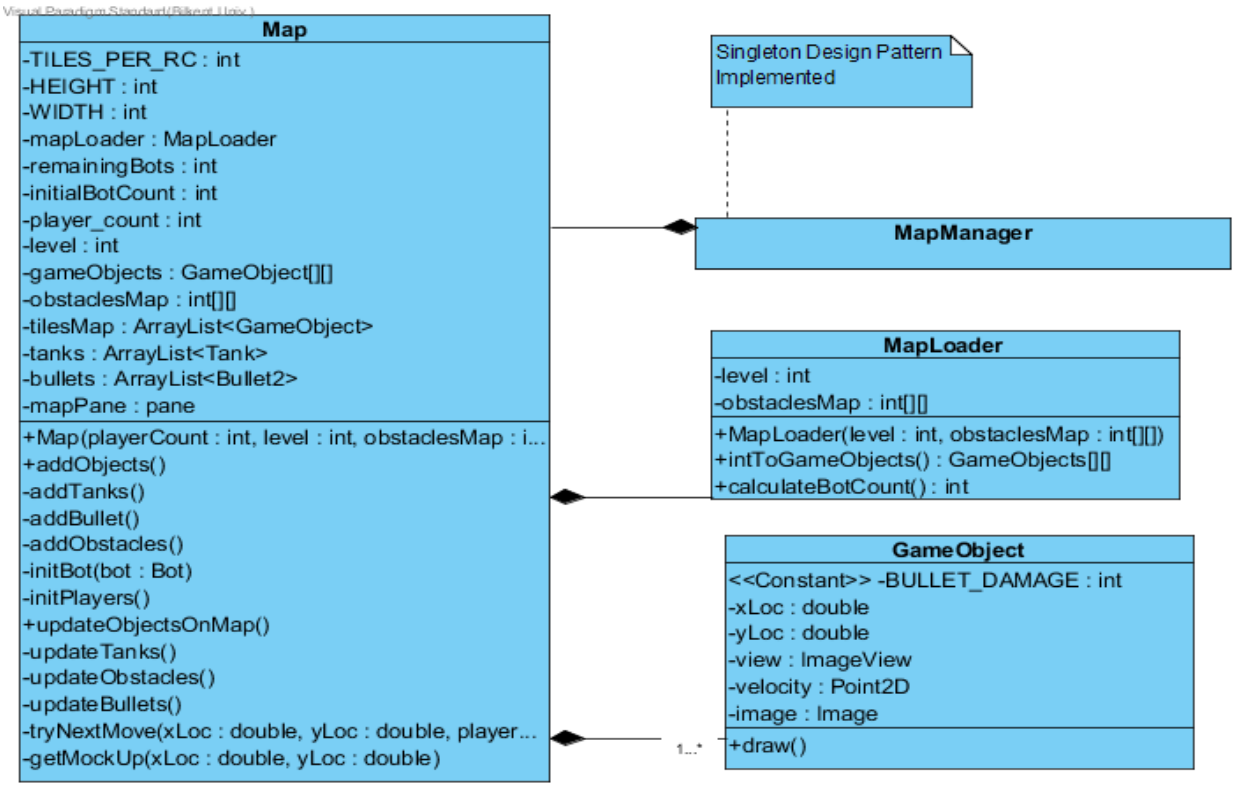

PauseMenu
-PAUSEMENU_WINDOW_WIDTH : int
-PAUSEMENU_WINDOW_HEIGHT : int
-vBox : VBox
-pauseMenuWindow : Stage
-pauseMenuScene : Scene
-returnCall : boolean
-returnButton : Button
-backToMenu : Button
-volumeBar : Slider
-volume : int
+PauseMenu()
+changeVolume(volume : int)
+addSettingsComponents()
<<Interface>>
ActionListener
<<Interface>>
KeyListener
<<Interface>>
KeyListener
<<Interface>>
MauseListener
Settings
-SETTINGS_WINDOW_WIDTH : int
-SETTINGS_WINDOW_HEIGHT : int
-settingsWindow : Stage
-settingsScene : Scene
-returnCall : boolean
-file : FileManager
-border : BorderPane
-player1Box : VBox
-player2Box : VBox
-soundBox : HBox
-buttonBox : HBox
-settings : Label
-menu : Menu
-player1Label : Label
-player1_keyList : ArrayList<String>
-player1_settings : ComboBox
-player1l : Label
-player2Label : Label
-player2_keyList : ArrayList<String>
-player2_settings : ComboBox
-submit : Button
-backToMenu : Button
-volume : int
-mute : CheckBox
-volumeBar : Slider
-musicBox : ComboBox<String>
-media : ArrayList<Media>
-player : MediaPlayer
-latestVolume : int
+Settings()
+addSettingsComponents()
+showSettings()
+mute()
+changeVolume()
+changeKeyList()
+updateUI()
+isReturnCall() : boolean
+update()
Menu
-MENU_BUTTON_COUNT : int
-MENU_WINDOW_WIDTH : int
-MENU_WINDOWS_HEIGHT : int
-menuWindow : Stage
-menuScene : Scene
-mBBox : VBox
-creditsFrame : ViewFrame
-howToPlayFrame : ViewFrame
-settings : Settings
-playerCount : int
-menuButtons : Button[]
-f : FileManager
-player : MediaPlayer
-battleCity : Label
+start(primaryStage : Stage)
+startSettings()
+showHowToPlay()
+showCredits()
+changeTheSong(song : String)
+startGame()
+stopSong()
+playSong()
+initMenuButtons(menuButtons : Button[])
+setPlayerCount(playerCount : int)
ViewFrame
-returnButton : JButton
-text : TextPanel
-title : Label
-frameImages : BufferedImage[]
-returnCall : boolean
+ViewFrame(title : String, text : String, images : Image)
+styleButton(returnButton : Button)
+isReturnCall() : boolean
ConfirmBox
-yesButton : Button
-noButton : Button
-answer : boolean
-answerLabel : Label
+display(title : String, message : String)
FileManager
<<Constant>> -TILES : int
<<Constant>> -NUMBER_IMAGES : int
<<Constant>> -NUMBER_AUDIOS : int
-curFil : File
-scannedImages : ArrayList<Image>
-scannedTexts : ArrayList<String>
-scannedAudios : ArrayList<Media>
-scan : Scanner
+FileManager()
+getMapLevelData(level : int) : int[][]
+saveHighestScore(score : int)
<<Interface>>
Serializer

Piece 2:

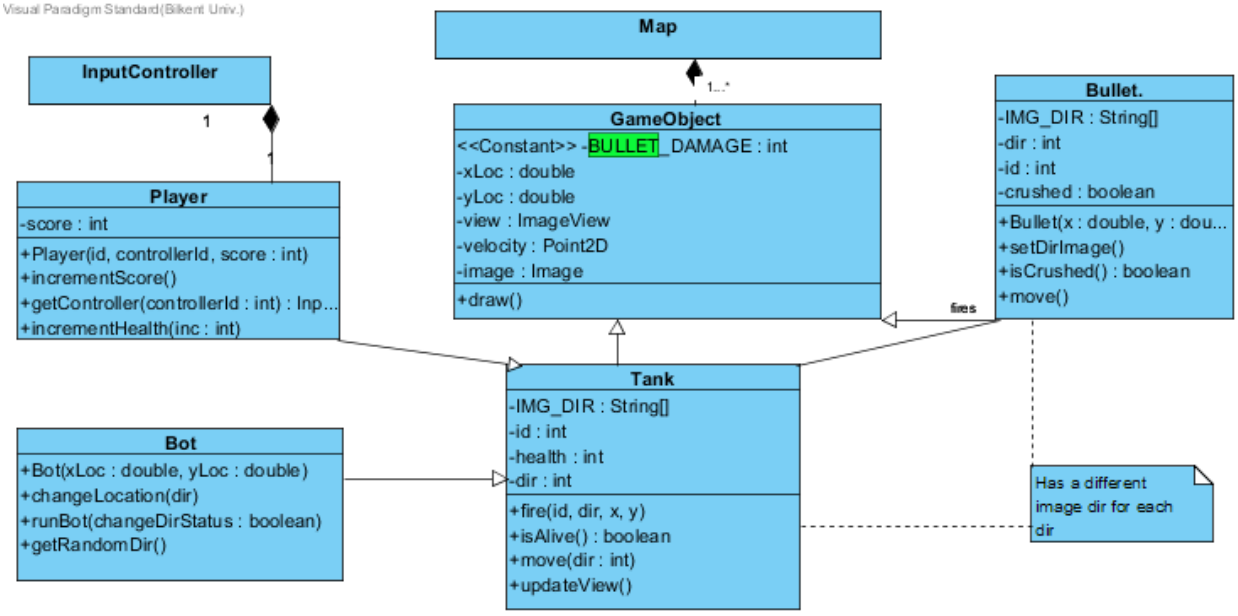
Note: For clarification reasons repeated classes collapsed in order to maintain briefty.



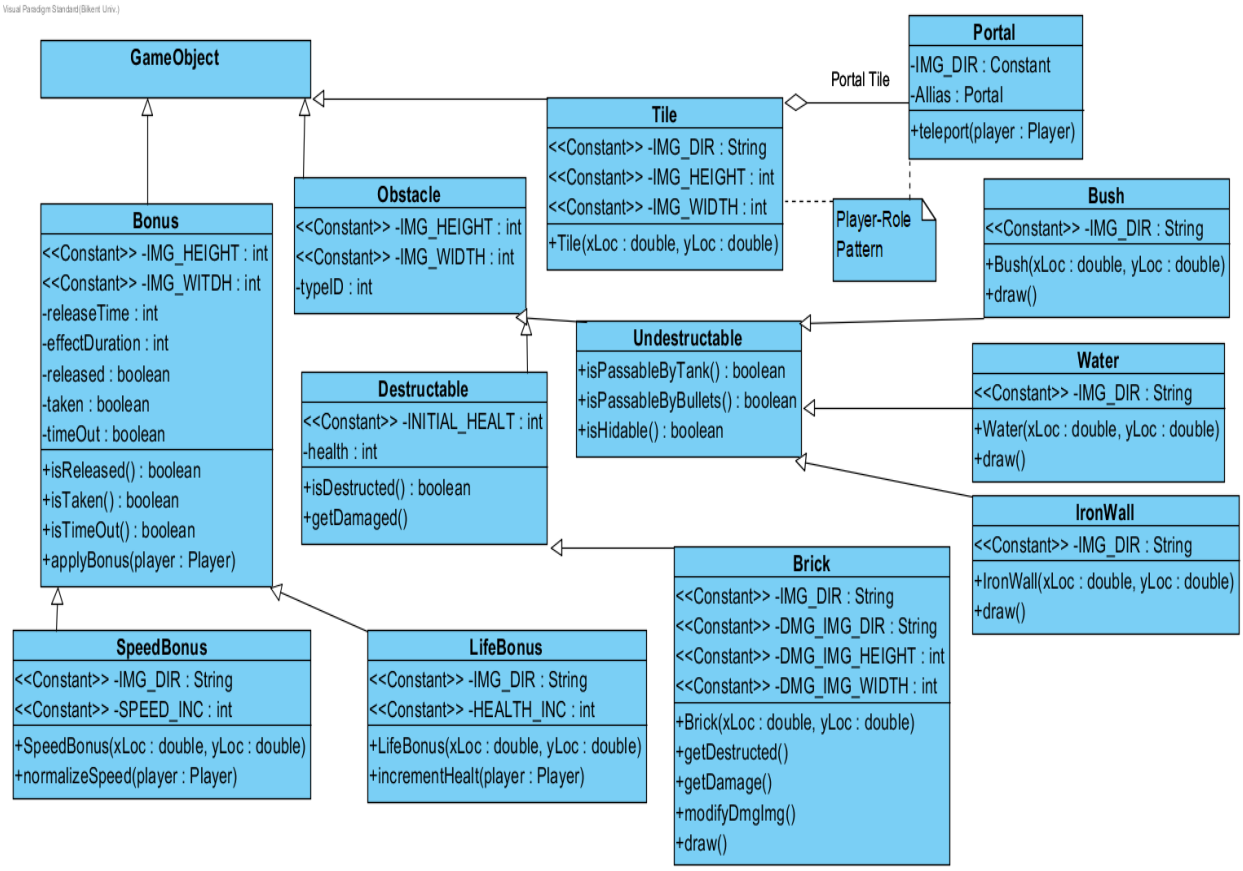
Piece 3:



Piece 4:



Piece 5:



4.2. Design Decisions-Design Patterns

During processes of low level design we have used some design pattern which will have their own tradeoffs.

4.2.1. Façade Design Pattern

To reduce the coupling between classes we have used **Façade design pattern** which is a structural design pattern. By using Façade pattern we have provided some interfaces to interact with the existing objects. For example, the only interaction between UserInterface subsystem and Management subsystem can occur through GameManagement and Menu class. These are the classes which provide interfaces for Façade design pattern.

However using Façade design pattern has its disadvantages, for example we cannot directly interact between menus and map management. Any change that must happen through menu to map must occur through Menu→GameManagement→MapManagement→Map, which requires many extra method calls and wastes unnecessary memory.

4.2.2. Singleton Design Pattern

To increase the maintainability of the game, we have applied singleton design pattern in some of the management classes (MapManager). Since there should be just one map during the game, we thought that a map with one than more manager can cause handling issues and errors during the game. That's why we decided to use only one single instance of MapManager class. By applying that, we have ensured that there will be only one modifier for the map at once.

However, by using we increased the coupling between MapManager, GameManager and Map and also unit testing become very hard for MapManager.

4.2.3. Player-Role Design Pattern

In our class diagram, it is possible to see that we have a Tile class that is associated with Portal role. Here we decided to use player-role design pattern, because we wanted Portal and a standart tile classes, which are the role classes of the Tile class to process independently and in different times when they are needed.

The Tile class is the player class which have two main independent roles in the game; Tile may be the Portal on the map or it can be a BackGround image.

This way we could also decrease the coupling, which will further be very handy to the game and its smooth processing.

4.3. Packages

In our implementation we have used two kinds of packages, first type is packages that are defined by us which includes our classes and second type is the packages that are introduced by external libraries such as JAVAFX.

4.3.1. Packages Introduced by Developers

4.3.1.1. Menu Package

Includes classes which are responsible for representation of the game menus and their methods.

4.3.1.2. Settings Package

Includes classes for settings and provides set of methods of changing and displaying settings.

4.3.1.3. File Management Package

Includes packages which provides file management in the game

4.3.1.4. Game Management Package

Includes classes which are responsible for game management, basically handles the flow between different components and subsystems.

4.3.1.5. Map Management Package

Includes classes which are responsible for map management uses Singleton design pattern to make the handling more reliable.

4.3.1.6. Game Entities Package

Includes classes which are the game objects of the game and provides methods to handle them.

4.3.2. External Library Packages

4.3.2.1. java.util

This package contain ArrayList which will be used as mentioned at classes to control multiple variables, such as control keys of a user, gameObjects currently on map, grid of the obstacles. Also Serializable in this package, which will be used for bytestreaming data from program to hard drive or vice versa. IO, subpackage will be used to interact with folders and .txt files, in order to import default maps.

4.3.2.2. javafx.scene.layout

This package provides User Interface layouts to fit GUI objects.

4.3.2.3. javafx.scene.paint

This package provides a set of classes to handle visual output.

4.3.2.4. javafx.animations

This package provides basic classes to controlling visual output, which is the main reason of our choice of javaFX, rather than java.

4.3.2.5. javafx.scene.events

This package provides the controll of the events. Delivery of inputs between different hardware components and exception handling is done by this package.

4.3.2.6. javafx.scene.input

This package provides input handlers for keyboard and mouse.

4.3.2.7. javafx.scene.image

This package provides the usage of images in program. Even basic obstacles with one color or two color-basic textures are easily drawable, we decided to use images of more complex shapes, such as tanks.

4.4. Class Interfaces

In this section, we have introduced our class interfaced more detailed and clear for any implementer to code it more efficiently.

4.4.1. Menu Class

Attributes:

- **private int MENU_BUTTON_COUNT:** This attribute will be used in the creation of the menuButtons array
- **private int MENU_WINDOW_WIDTH:** This attribute will be used in creation of the Menu frame
- **private int MENU_WINDOWS_HEIGHT:** This attribute will be used in creation of the Menu frame
- **private Stage menuWindow:** This attribute will be used in creation of the Menu Window
- **private Scene menuScene:** This attribute will be used in creation of the Menu Scene
- **private VBox mBBox:** This attribute will be used as a box for the buttons on the Menu frame
- **private ViewFrame creditsFrame:** This attribute will be used for creating the Credits frame when the “Credits” button is clicked
- **private ViewFrame howToPlayFrame:** This attribute will be used for creating the How To Play frame when the “How To Play” button is clicked
- **private Settings settings:** This attribute will be used for creating the Settings frame when the “Settings” button is clicked
- **private int playerCount:** This attribute will be used to determine the count of the player (depending on whether single or multiplayer game mode button was clicked)
- **private Button[] menuButtons:** This attribute will be used in creation of the menu buttons on the Menu frame
- **private FileManager f:** This attribute will be used in order to pass the data of Credits and How To Play screens to ViewFrame class and to access the background song for MediaPlayer
- **private MediaPlayer player:** This attribute will be used in order to play the background song

- **private Label battleCity:** This attribute will be used to display the name of the game on Menu frame

Constructors:

- **public start(primaryStage : Stage):** Default constructor for the Menu class.

Methods:

- **private void startSettings():** This method will be used to open the Settings frame when the “Settings” button is clicked
- **private void showHowToPlay():** This method will be used to open the How to Play frame when the “How To Play” button is clicked
- **private void showCredits():** This method will be used to open the Credits frame when the “Credits” button is clicked
- **public void changeTheSong(song : String):** This method will be used to change the background song chosen from the Settings class
- **private void startGame():** This method will be used to start the game when any of the game buttons (Singleplayer or Multiplayer) is clicked
- **public void stopSong():** This method will stop the background song if the mute checkbox was clicked in the Settings frame
- **public void playSong():** This method will play the background song if the mute checkbox was unclicked in the Settings frame
- **private void initMenuButtons(menuButtons : Button[]):** This method will initialize the buttons and style them for the interface of Menu frame
- **private setPlayerCount(playerCount :** int): This method will set the count of the players depending on which game button is clicked

4.4.2. PauseMenu Class

Attributes:

- **private int PAUSE_WINDOW_WIDTH:** This attribute will be used in the creation of the PauseMenu frame
- **private int PAUSEMENU_WINDOW_HEIGHT:** This attribute will be used in the creation of the PauseMenu frame
- **private int VBox vBox:** This attribute will be used to hold the buttons of the PauseMenu frame in the box for the interface
- **private Stage pauseMenuWindow:** This attribute will be used for creating the PauseMenu Window
- **private Scene pauseMenuScene:** This attribute will be used for creating the PauseMenu Scene
- **private boolean returnCall:** This attribute will be used for determining whether or not the user returned to the game or not
- **private Button returnButton:** This attribute will be used to create a “return” button on PauseMenu frame, which will return user to the game
- **private Button backToMenu:** This attribute will be used to create a “back” button on PauseMenu frame, which will create Menu frame and finish the game
- **private Slider volumeBar:** This attribute will be used to create the volume bar on PauseMenu frame, which will help to adjust the background song volume from the PauseMenu frame
- **private int volume:** This attribute will be used to keep track of the background song’s volume

Constructors:

- **public PauseMenu():** Default constructor for the PauseMenu class.

Methods:

- **private void changeVolume(volume : int):** This method will change the volume depending on the value of volumeBar

- **private void initSlider():** This method will initialize the volumeBar and give it a style

- **private void addSettingsComponents():** This method will add all the components on the PauseMenu frame for the interface

4.4.3. ViewFrame Class

Attributes:

- **private JButton returnButton:** This attribute provides a button in the frame which has a function to return the screen to previous frame. (E.g. In the how to play screen you can return the main menu by pressing the returnButton)
- **private JTextPanel text:** This attribute proves text field for the frame. The necessary texts will be taken by fileManager class and integrated into necessary frames.

- **private JLabel title:** This attribute provides title for the frame.(E.g. “How to Play”, “Credits” …)
- **private BufferedImage[] frameImages:** This array attribute keeps the images in the array so that when the frame is constructed. It takes them initializes on the screen.

- **private boolean returnCall:** This attribute will check whether or not the ViewFrame class was accessed and will be used in Menu class

Constructors:

- **public ViewFrame():** Default constructor for ViewFrame class. Basically initializes an empty Frame to be filled in.
- **public ViewFrame(title : String, text : String, images : Bufferedİmage[]):** This constructor takes string title, string text and buffered image array images of a Frame and initializes a ViewFrame.

Methods:

- **private void styleButton(returnButton : Button):** This method will initialize the returnButton and give it a style
- **private boolean isReturnCall():** This method will return true if the ViewFrame class was accessed

4.4.4. Settings Class

Attributes:

- **private int SETTINGS_WINDOW_WIDTH:** This attribute will be used in the creation of Settings frame
- **private int SETTINGS_WINDOW_HEIGHT:** This attribute will be used in the creation of Settings frame
- **private Stage settingsWindow:** This attribute will be used in the creation of Settings Window
- **private Scene settingsScene:** This attribute will be used in the creation of Settings Scene
- **private boolean returnCall:** This attribute will be used to check whether or not Settings frame is accessed
- **private FileManager file:** This attribute will be used to load the background songs for the musicBox
- **private BorderPane border:** This attribute will be used to create a border on the Settings frame for a better alignment of other attributes
- **private VBox player1Box:** This attribute will be used to gather player1Label, player1_keyList and player1_settings in one vertical area
- **private VBox player2Box:** This attribute will be used to gather player2Label, player2_keyList and player2_settings in one vertical area
- **private HBox soundBox:** This attribute will be used to gather volumeBar, mute and musicBox in one horizontal area
- **private HBox buttonBox:** This attribute will be used to gather submit and backToMenu buttons in one horizontal area
- **private Label settings:** This attribute will be used to display the “Settings” on the Settings frame

- **Menu menu:** This attribute is the instance of Menu class and will be used to access the methods of Menu class in order to play, change or stop the song
- **private Label player1Label:** This attribute will be used to display the “Player 1” on the Settings frame
- **private ArrayList<String> player1_keyList:** This attribute will keep the list of sets of control keys of the first player
- **private ComboBox player1_settings:** This attribute will display the content of player1_keyList so that the user would be able to choose the input keys set
- **private Label player2Label:** This attribute will be used to display the “Player 2” on the Settings frame
- **private ArrayList<String> player2_keyList:** This attribute will keep the list of sets of control keys of the second player
- **private ComboBox player2_settings:** This attribute will display the content of player2_keyList so that the user would be able to choose the input keys set
- **private Button submit:** This attribute will be the “Submit” button on the Settings frame
- **private Button backToMenu:** This attribute will be the “Back” button on the Settings frame
- **private int volume:** This attribute will keep track of the volume of the background song
- **private CheckBox mute:** This attribute will be the mute checkbox on the Settings frame
- **private Slider volumeBar:** This attribute will be the volumeBar slider on the Settings frame
- **private ComboBox<String> musicBox:** This attribute will be the list of songs for choosing on the Settings frame
- **private ArrayList<Media> media:** This attribute will hold the name of background songs in the ArrayList
- **private MediaPlayer player:** This attribute will be the player for playing the background song
- **private int latestVolume:** This attribute will keep track of the old volume in order to display it after the unmute() function

Constructors:

- **public Settings():** This constructor will be the default constructor of Settings class

Methods:

- **private void addSettingsComponents():** This method will add all the components to the Settings frame
- **public void showSettings():** This method will display the Settings frame
- **private void selectBackGroundMusic(song : String):** This method will be used in order to select a new background song from the musicBox
- **private void initCheckBox():** This method will initialize and style mute
- **private void initSlider():** This method will initialize and style volumeBar
- **private void initButtons():** This method will initialize and style backToMenu and submit
- **private void initComboBoxes():** This method will initialize and style musicBox, player1_keyList and player2_keyList
- **private void initPlayerLablels():** This method will initialize and style player1Label and player2Label
- **private void mute():** This method will mute the background song
- **private void unmute():** This method will unmute the background song
- **private void changeVolume():** This method will adjust the background song depending on the value on volumeBar
- **private void changeKeyList():** This method will change the control keys of the players
- **private void updateUI():** This method will update the Settings frame after choosing the new set from the ComboBoxes
- **public boolean isReturnCall():** This method will return true if the Settings class was accessed

- **private void update():** This method will update the changed settings after the "Submit" button was clicked

4.4.5. InputController Class

Attributes:

- **private Player player:** This attribute will keep track of the first or second player's inputs
- **private Scene mapScene:** This attribute will update the location of the player (first or second) on the Map
- **private Map map:** This attribute is the instance of the Map class and will be used to update the locations of the player tanks on the Map

Constructors:

- **InputController(mapManager: MapManager, player : Player):** Constructor with the parameters, which creates the InputController frame, determines the player id and the player inputs

Methods:

- **public void handle(e : KeyEvent):** This method will handle the player inputs with the KeyEvent
- **public void movePlayer():** This method will update the player's location on the Map
- **public void firePlayer():** This method will update the Map state if the fire is made

4.4.6. ConfirmBox class

Attributes:

- **private Button yesButton:** This attribute will be the "Yes" button on the ConfirmBox frame
- **private Button noButton:** This attribute will be the "No" button on the ConfirmBox frame
- **private boolean answer:** This attribute will be the reply of the user depending on which button he/she clicked

- **private Label answerLabel:** This attribute will be the label for displaying on the ConfirmBox frame

Methods:

- **private void display(title : String, message : String):** This method will display the message passed from the Menu class
- **private void styleButtons(yes: Button, no: Button):** This method will initialize and style the yesButton and noButton

4.4.7. FileManager class

Attributes:

- **private final int TILES:** This attribute will be the number of tiles which will be used to generate the map for the game
- **private final int NUMBER_IMAGES:** This attribute will be the number of images to be used for generating the game entities on the map for the game
- **private final int NUMBER_AUDIOS:** This attribute will be the number of audio files to be used for the background songs
- **private File curFil:** This attribute is the instance of the File class and will be used to scan files
- **private ArrayList<Image> scannedImages:** This attribute will be the arraylist for holding the images in it
- **private ArrayList<String> scannedTexts:** This attribute will be the arraylist for holding the data for ViewFrame class
- **private ArrayList<Media> scannedAudios:** This attribute will be the arraylist for holding the audio files in it.
- **private Scanner scan:** This attribute will be used to scan the files and upload them to the corresponding arraylist

Constructors:

- **public FileManager():** This is the default constructor of FileManager class

Methods:

- **private int[][] getMapLevelData(level : int):** This method will return the file for each level, which will be used in creation of the according map
- **private void saveHighestScore(score: int):** This method will save the highest score in the game
- **private int getHighestScore():** This method will return the highest score of the game
- **private ArrayList<String> getHowToPlayDoc():** This method will scan the file for How to Play screen and add the data to the arraylist by each line
- **private ArrayList<String> getSettingsDoc():** This method will scan the file for Settings screen and add the data to the arraylist by each line
- **private ArrayList<Image> getScannedImages():** This method will scan the images and add them to scannedImages
- **private ArrayList<Media> getScannedAudios():** This method will scan the images and add them to scannedAudios
- **private Media getOpeningSong():** This method will return the song for the opening menu
- **private Media getGeneralSong():** This method will return the song for the other screens of the game

4.4.8. MapManager class

MapManager class control the game loop through **AnimationTimer**.

Attributes:

- **private MapManager mapManager:** This attribute is the instance of the MapManager class
- **private boolean gameStatus:** This attribute will be used to keep track of the game status
- **private int mapLevel:** This attribute will be used to keep track of the map level

- **private Map map:** This attribute is the instance of the Map class and will be used to generate the Map with corresponding mapLevel
- **private boolean mapFinished:** This attribute will be used to check if the map generation is finished
- **private FileManager mapManagerFileManager:** This attribute will be used to scan the corresponding map details
- **private int[][] obstaclesMap:** This attribute will be used to hold the obstacles
- **private CollisionManager collisionManager:** This attribute is the instance of the CollisionManager class and will be used to keep track of collisions
- **private AnimationTimer timer:** This attribute is the instance of the AnimationTimer class and will be used to create a timer for bot tanks

Constructors:

- **public MapManager():** This is the default constructor of MapManager class

Methods:

- **private void createMapManager():** This method will create the MapManager
- **private void setProperties(level : int, playerCount: int):** This method will be used to set the level and playerCount for the game
- **private MapManager getMapManager():** This method will return the created MapManager instance
- **private void start(stage: Stage):** This method is working as the helper constructor
- **private void onUpdate():** This method will be used to update the map
- **private void updateAllObjects():** This method will be used to update the objects on the map
- **private void checkBots():** This method will be used to check if there are bots on the map
- **private void startLevel():** This method will be used to start the level

- **private void handleBots():** This method will be used to create the bots
- **private void readObstaclesMap(level :** int): This method will be used to read the obstacles from the file and upload them to the map
- **private void stopGameLoop():** This method will be used to stop the game loop
- **private void gameLoop():** This method will be used to start the game loop
- **private boolean isMapFinished(completedLevels : int):** This method will return true if the map is finished generating
- **private void finishLevel():**This method will be used to finish the level

4.4.9. Map class

Attributes:

- **private int TILES_PER_RC:** This attribute will be the number of tiles per rectangle
- **private int HEIGHT:** This attribute will be the size of the height of the map
- **private int WIDTH:** This attribute will be the size of the width of the map
- **private MapLoader mapLoader:** This attribute is the instance of the MapLoader class and will be the helper of the Map class
- **private int remainingBots:** This attribute will be the number of the remaining bots
- **private int initalBotCount:** This attribute will be the number of the bots in the start of the game
- **private int player_count:** This attribute will be the count of the players depending on the game mode
- **private int level:** This attribute will be the number of the level the player is on
- **private GameObjects[][] gameObjects:** This attribute will hold the game objects inside
- **private int[][] obstaclesMap:** This attribute will hold the obstacles inside
- **private ArrayList<GameObject> tilesMap:** This attribute will hold the tiles inside

- **private ArrayList<Tank> tanks:** This attribute will hold the tank objects inside
- **private ArrayList<Bullet> bullets:** This attribute will hold the bullet objects inside
- **private Pane mapPane:** This attribute will be used to create a pane for the map

Constructors:

- **public Map(playerCount : int, level : int, obstaclesMap : int[][]):** This constructor is the constructor with parameters which will create the map

Methods:

- **private void addObjects():** This method will add all the objects to the map
- **private void addTanks():** This method will add the tanks to the map
- **private void addBullet():** This method will add the bullet objects to the map
- **private addObstacles():** This method will add the obstacles to the map
- **private void initBot(bot : Bot):** This method will initialize the bots
- **private void initPlayers():** This method will initialize the player tanks
- **private void updateObjectsOnMap():** This method will update the objects on the map
- **private void updateTanks():** This method will update the tanks on the map
- **private void updateObstacles():** This method will update the obstacles on the map
- **private void updateBullets():** This method will update the bullet objects on the map
- **private void tryNextMove(xLoc : double, yLoc : double, playerViw : ImageVoew):** This method will update the map by each move
- **private void getMockUp(xLoc : double, yLoc : double):** This method will update the position of the objects

4.4.10. MapLoader class

Attributes:

- **private int level:** This attribute will be number of the level
- **private int[][] obstaclesMap:** This attribute will hold the obstacles inside

Constructors:

- **public MapLoader(level : int, obstaclesMap : int[][]):** This is the constructor with the parameters which will load the map

Methods:

- **private GameObjects[][] intToGameObjects():** This method will convert the numbers from the scanned file to the corresponding game objects and return them in the 2D array
- **private int calculateBotCount():** This method will calculate and return the bot count

4.4.11. GameManager class

Attributes:

- **private final int GAME_START_LEVEL:** This attribute provides default level count for the game, which is going to initialize as 1.
- **private final int GAME_FINAL_LEVEL:** This attribute provides the final level of the game
- **private MapManager mapManager:** This attribute is the instance of the MapManager class and provides the linkage between the logic of the game and map display
- **private FileManager gameManagerFileManager**: This attribute is the instance of the FileManager class and provides the GameManager with the files from the FileManager
- **private InputController inputController**: This attribute is the instance of the InputController class and controls the user inputs during the game itself
- **private int highestScore**: This attribute keeps track of the highest score in the game
- **private Player[] players**: This attribute is an array of the Player class and aims to create the players with the identification number adjustment for each

- **private int[] currentScores**: This attribute is an integer array which keeps track of the scores for each player depending on their identification number
- **private boolean gameRunning**: This attribute keeps track of whether the game is running or not
- **private boolean gamePaused**: This attribute keeps track of whether the game is paused or not
- **private int currentGameLevel**: This attribute provides the game with the number of the current level the player is on, or passed to
- **private boolean gameFinished**: This attribute keeps track of whether the game is finished or not
- **private Button pauseButton:** This attribute will be the “Pause” button on the Game screen
- **private PauseMenu pause:** This attribute will be used to create the PauseMenu screen when the “Pause” button is clicked

Constructors:

- **GameManager():** Default constructor for the GameManager class, which simply initializes the class

Methods:

- **private boolean isGameOver():** This method checks whether the game is over or not
- **private void updateCurrentScore(player_id : int):** This method updates the score of the particular player depending on the player_id
- **private void updateHighestScore(highestScore : int):** This method updates the highest score of the game
- **private void startGame():** This method simply starts the game
- **private void finishGame():** This method simply finishes the game

- **private boolean isGameRunning():** This method checks whether the game is currently running or not
- **private boolean isGamePaused():** This method checks whether the game is currently paused or not
- **private void pauseGame():** This method simply pauses the game
- **private void continueGame():** This method simply continues the game, if it was paused before
- **private void quitGame():** This method simply quits the game
- **private void saveHighestScore(scoregame: int):** This method saves the highest score of the game

  private int getHighestScore(): This methos provides the game with the highest score by returning it
- **private boolean isGameFinished():** This method checks whether the game is finished or not
- **private void startLevel(mapManager: MapManager, nextLevel: int):** This method starts the next level, if the user passed it or finished the level
- **private void finishLevel(mapManager: MapManager):** This method finishes the level.

4.4.12. CollisionManager class

Attributes:

- **private ArrayList<Tank> tanks:** This attribute will hold the tanks inside
- **private ArrayList<Bullet> bullets:** This attribute will hold the bullet objects inside
- **private ArrayList<GameObject> gameObjects:** This attribute will hold the game objects inside

Constructors:

- **CollusionManager(gameObjects : ArrayList<GameObject>, bullets : ArrayList<Bullet>, tanks : ArrayList<Tanks>):** This is the constructor with parameters which will check the collisions

Methods:

- **public void checkCollision():** This method will implement the process of collision.
- **public void updateRemovals():** This method will update the map if there are some removals on the map
- **public void checkObstacleCollision():** This method will check if there are the obstacle collisions

4.4.13. GameObject class

Attributes:

- **private final int BULLET_DAMAGE:** This attribute will provide the game with the number of bullet damages
- **private double xLoc:** This attribute holds the left point of the object.
- **private double yLoc:** This attribute holds the top point of the object.
- **private ImageView view:** This attribute will display the objects on the map
- **private Image image:** This attribute holds image file for objects.
- **private Point2D velocity:** This attribute will keep track of the location of the tanks and give them velocity

Methods:

- **public void draw():** This method draws image of the object in pixels calculated by position and size of the object.

4.4.14. Tank class

Attributes:

- **private String[] IMG_DIR:** This attribute will hold all the possible images of the player's tank
- **private int id:** This attribute holds the id of the tank, which determines the owner of the bullets.
- **private int health:** This attribute holds the current health of the tank.
- **private int dir:** This attribute determines the direction of the bullet and image(front of tank).

Methods:

- **public void fire(id: int, dir: int, x: int, y: int):** Creates a bullet object when the tank is firing
- **public boolean isAlive():** This method determines whether the tank is dead.
- **public void move(dir: int):** This method requests the movement on map.
    - **public void updateView():** This method updates the view of the tank when it moves

4.4.15. Player class

Attributes:

- **private int score:** This attribute holds the score of the player, which is incremented by killing bots and finishing levels.

Constructors:

- **Player(id: int, controllerId: int, socre : int):** The constructor of player class determines ID of the players and their scores

Methods:

- **public void incrementScore():** This method updates the score of player when a level finishes or player kills a bot.
- **public InputController getController(controllerId: int):** This method return the controller input keys based on player's id.
- **public void incrementHealth(inc : int):** This method increments the health of the player when the bonus was taken

4.4.16. Bot class

Constructors:

- **public Bot(xLoc : double, yLoc : double):** This constructor creates the bot on the map on the specified location

Methods:

- **private changeLocation(dir : int):** This method changes the location of the bot
- **private runBot(changeDirStatus : boolean):** This method moves the bot with the direction initialized before
- **private void getRandomDir():** This method creates new random direction for the bot tanks

4.4.17. Bullet class

Attributes:

- **private String[] IMG_DIR:** This attribute holds all possible images of the bullet inside
- **private int dir:** This attribute provides the class with bullet's direction
- **private int id:** This attribute will be used to determine to which tank the bullet belongs

- **private boolean crushed:** This attribute will check if the bullet hit another object and disappear

Constructors:

- **Bullet(x: double, y: double):** Constructor of the bullet. Creates a bullet from the given position through to given direction.

Methods:

- **private void setDirImage():** This method sets the image for corresponding direction of the bullet
- **private boolean isCrushed():** This method returns true if the bullet hit something and disappeared
- **private void move():** This method will be used to move the bullet object

4.4.18. Destructable class

Attributes:

- **private final int INITIAL_HEALTH:** This attribute holds the initial health of the destructible object.
- **Private int health:** This attribute holds the health of the destructible object

Methods:

- **public boolean isDestructed():** This method determines if the health of the object is less than or equal to 0.
- **public void getDamaged():** This method processes when the obstacle is getting damaged

4.4.19. Obstacle class

Attributes:

- **private final int IMG_HEIGHT:** This attribute holds the height for the obstacle images
- **private final int IMG_WIDTH:** This attribute holds the width for the obstacle images
- **private int typeID:** This attribute determines the type of the obstacle

4.4.20. Undestructable class

Methods:

- **public boolean isPassableByTank():** If the obstacle is bush, this method returns true.
- **public boolean isPassableByBullets():** If the obstacle is water, this method returns true.
- **public boolean isHideable():** If the obstacle is bush, this method returns true.

4.4.21. Tile class

Attributes:

- **public final String IMG_DIR:** This attribute provides the class with the tile image
- **public final int IMG_HEIGHT:** This attribute gives the image height
- **public final int IMG_WIDTH:** This attribute gives the image width

Constructors:

- **public Tile(xLoc : double, yLoc : double):** This constructor creates tile in specified position

4.4.22. Bonus class

Attributes:

- **public final int IMG_HEIGHT:** This attribute gives the image height
- **public final int IMG_WIDTH:** This attribute gives the image width
- **private int releaseTime:** This attribute will keep the time of the bonus release
- **private int effectDuration:** This attribute will keep the time of the effect period of the bonus
- **private boolean released:** This attribute will check whether or not the bonus is released
- **private boolean taken:** This attribute will check if the bonus is taken by the tank
- **private boolean timeOut:** This attribute will check if the bonus should disappear

Methods:

- **public boolean isReleased():** This method returns true if the bonus is released
- **public boolean isTaken():** This method returns true if the bonus is taken by a tank
- **public boolean isTimeOut():** This method returns true if the bonus should disappear
- **public void applyBonus(player : Player):** This method applies the bonus to the specific player

4.4.23. LifeBonus class

Attributes:

- **public final String IMG_DIR:** This attribute gives the image direction
- **public final int HEALTH_INC:** This attribute gives the constant life increment for the tanks

Constructors:

- **public LifeBonus(xLoc : double, yLoc : double):** This constructor releases the life bonus in specified location

- **public void incrementHeal(Player player):** This method increments the health of the player who takes the health bonus.

4.4.24. SpeedBonus class

Attributes:

- **public final String IMG_DIR:** This attribute gives the image direction
- **public final int SPEED_INC:** This attribute gives the constant speed increment for the tanks

Constructors:

- **public SpeedBonus(xLoc : double, yLoc : double):** This constructor releases the speed bonus in specified location

Methods:

- **public void normalizeSpeed(player : Player):** This method normalizes the speed of the tank after the period of effect was passed

4.4.25. Portal Class

Attributes:

- **public final String IMG_DIR:** This attribute gives the image direction
- **public Portal Allias:** This attribute will be the portal object on the map

Methods:

- **public void teleport(player:Player):** Teleports the player to the allias portal.

4.4.26. Brick Class

Attributes:

- **public final String IMG_DIR:** This attribute gives the image direction
- **public final String DMG_IMG_DIR:** This attribute gives the damaged brick image the direction
- **public final int DMG_IMG_HEIGHT:** This attribute gives the image height
- **public final int DMG_IMG_WIDTH:** This attribute gives the image width

Constructors:

- **public Brick(xLoc : double, yLoc : double):** This constructor creates the object in specified location

Methods:

- **public void getDestructed():** This method destroys the obstacle

- **public void getDamage():** This method damages the obstacle
- **public void modifyDmgImg():** This method updates the image of the obstacle
    - **public void draw():** Overrides the method from the GameObject class

4.4.27. Bush Class

Attributes:

- **public final String IMG_DIR:** This attribute gives the image direction

Constructors:

- **public Bush(xLoc : double, yLoc : double):** This constructor creates the object in specified location

Methods:

- **public void draw():** Overrides the method from the GameObject class

4.4.28. Water Class

Attributes:

- **public final String IMG_DIR:** This attribute gives the image direction

Constructors:

- **public Water(xLoc : double, yLoc : double):** This constructor creates the object in specified location

Methods:

- **public void draw():** Overrides the method from the GameObject class

4.4.29. IronWall Class

Attributes:

- **public final String IMG_DIR:** This attribute gives the image direction

Constructors:

- **public IronWall(xLoc : double, yLoc : double):** This constructor creates the object in specified location

Methods:

- **public void draw():** Overrides the method from the GameObject class.

5. Improvement Summary

In the second iteration of the design report we have done significant changes. Firstly, because of the additions (portal, bonuses) we did in the second analysis report, our objects design has changed. Also, because of other changes we did during design like adding some extra classes like MapLoader to partition the workload of MapManager has also affected our object design. Since our object design is affected, accordingly we have added some additional trade-offs for high level design and also for object design trade-offs. We will list the important changes for briefty of the report:

- We have completely changed our system decomposition. Our main design architectural pattern which is MVC is stayed as the same. However, our subsystems have changed. In the second iteration rather than writing all of the classes in a subsystem. We have **decomposed** our system in **three** main components UserInterface, Management, Model which are equivalent to View, Controller, Model respectively.
- Our representation of subsystem decomposition is totally changed. To show the relationship between subsystems we have used **lollipop component diagrams**. Briefly, there is **two way association** between Controller and View, **one way association** between Controller and Model(Controller to Model) and **there is no direct association** between Model and View.
- We have decomposed each subsystem to smaller components if possible. The division of components made according to functionalities of different classes. We have changed the names of subsystem that we used in the first iteration.
    - We have divided UserInterface subsystem to 3 smaller components(Menu, Frame, Settings). Their functionality explained in above.
    - We have divided Management subsystem to 4 smaller components. Their functionality explained in above.
    - The Model system has only one component which is GameEntities for now.
- We have added some new classes like Bonus(and its children), Tile(and its child), MapLoader…

- We have deleted some classes because their presence found unnecessary. E.g. Screen Manager.
- In Subsystem decomposition, we explained our subsystems and their relationship with our architectural design pattern which is MVC.
- In subsystem interfaces we have explained the subsystems more detailed and also we gave important information about their components.
- We have added some **design patterns** to achieve to have a complete object oriented design. Eg. **Singleton Design Pattern, Façade Design Pattern and Player-Role Pattern.**
- We have added **'Design Patterns' section** in which we have discussed our design patterns more detailed and explained their trade-offs.
- In the low level design, the object class diagram is updated. Its complete version and some pieces are given for more clarity on object model.
- The class interfaces are updated. Class explanations made as clear as possible.
- The packages are updated.

6. Contributions in Second Iteration

All Sections revised by Kaan Sancak, Mahin Khankishizade.

Trade-offs revised by Ozan Kerem Devamlı.