CS 319 - Object-Oriented Software Engineering
Project Design Report

Battle City

Group 1-I
Mert Aytöre
Mert Özerdem
Muhammed Yusuf Satıcı
Mehmet Orçun Yalçın

1. Introduction

1.1.Purpose of the System

Battle City is a basic 2-D strategy game aiming to entertain users while they are reaching their opponent's base by destroying the bricks. It is influenced by the legendary Atari game "Tank 1990" with some different and enjoyable extra features. By adding three different kinds of maps with different difficulty levels and various objects that users can use during the game, our game is distinguishable among other its ancestors. Battle City aims to make game more interesting and more challenging.

1.2.Design Goals

1.2.1. Adaptability

The game "Battle City" will be implemented with Java because it provides cross-platform portability contrary to C or C++. This feature enables our game to work in all JRE installed platforms. So users will be able to play the "Battle City" in any platform. Additionally, swing framework will be used to handle graphics. We are decided to define every part of the system explicitly in order to represent our Object Oriented approach.

1.2.2. Efficiency

In our games, we aim to make the effects, each movement of every object, animations very smoothly. In briefly, main purpose of our system is fluent gameplay. It is critical that users' satisfaction and their requests should be responded in an instant, for this purpose the game can respond the users within only one second. In this way any delay can't distract the users' entertainment. Moreover audio files that have .wav and .mp3 extensions will be used so the sounds of the game don't affect the game's performance.

1.2.3. Usability

Tank games are one of the most loveable Atari games all around the world for all of the ages so our remake game aims to make a game that can be played by anyone. Simplicity in the usage is a crucial design goal so our game promises simple and user-friendly interfaces to the user. Therefore, although our game shall not contain complicated connections or definitions that discomfort the user, our effective menu shall help users to travel all features of games easily.

1.2.4. Reliability

Our system will be bug free and will be arranged to not crush with irrelevant inputs or when the users choose invalid actions. To achieve this goal, the tests will be done for each part continuously. Furthermore any information of the previous games such as names, high scores of the players will be preserved and users certainly shall not change.

1.2.5. Modifiability

In our system, the objects and maps shall be modifiable and open to new features. Also, users shall modify keyboard settings in order to make it convenient.

1.3.Definitions, acronyms and abbreviations

MVC: Model view controller

JDK: Java Development Kit

JRE: Java Runtime Environment

1.4.References

2. Software Architecture

2.1.Overview

In this subsection of the report the decomposition of the subsystem to the subsystems is shown. The layers and dependencies between the subsystems are explained and the hardware and software components are given. Also, the boundaries for the entities and interfaces are discussed along with the object's lifecycle. The authentication, authorization and data management processes are explained in details.

In the Battle City software project, the Model/View/Controller will be used as the architectural style since it enables us to divide the system into subsystems in accordance with the boundary, control and entity objects. Also, it provide fast access to the model since the model subsystem do not depend on the data flow of controller or view subsystems.

2.2.Subsystem Decomposition

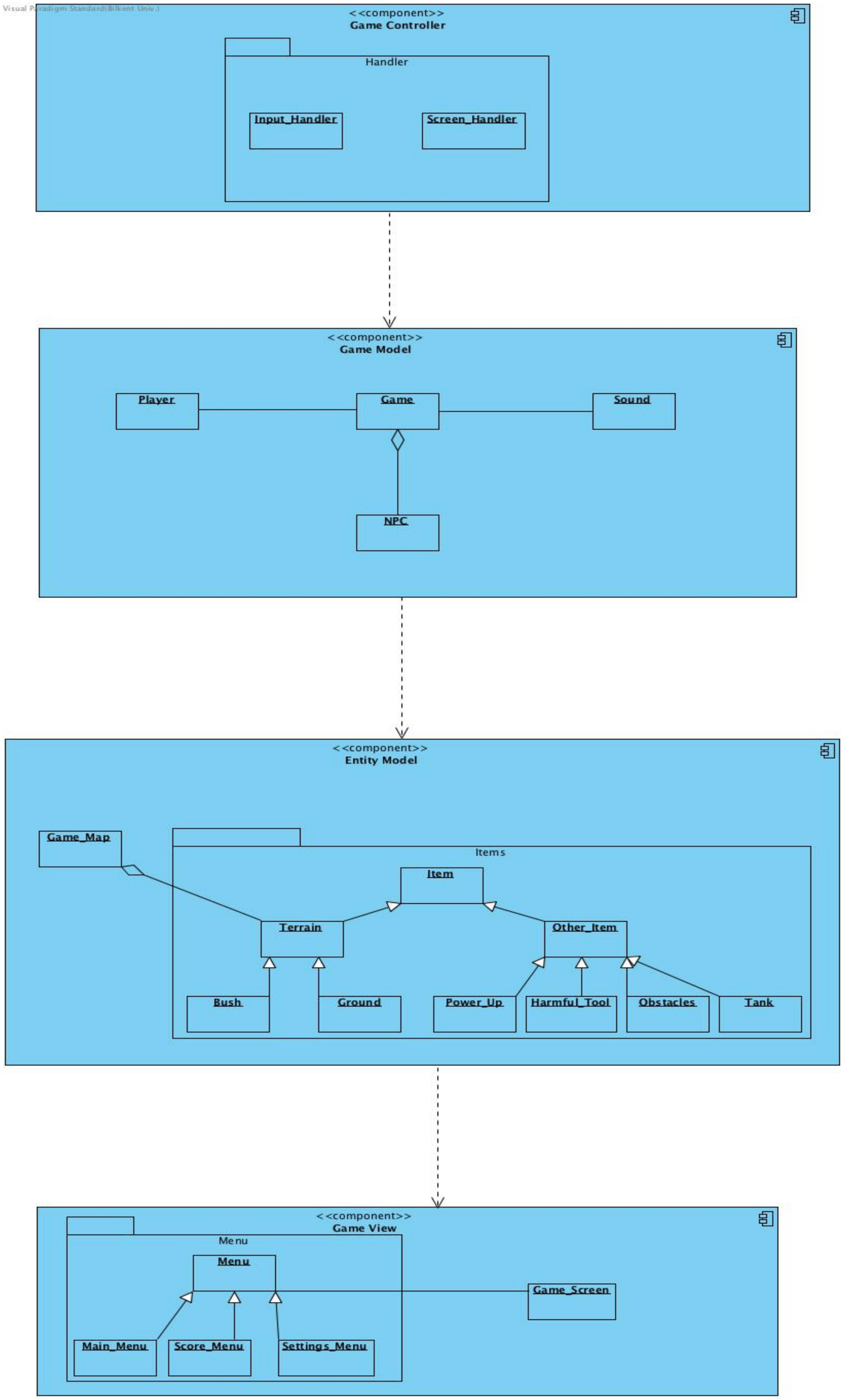


Figure-1 Subsystem Decomposition with Dependencies

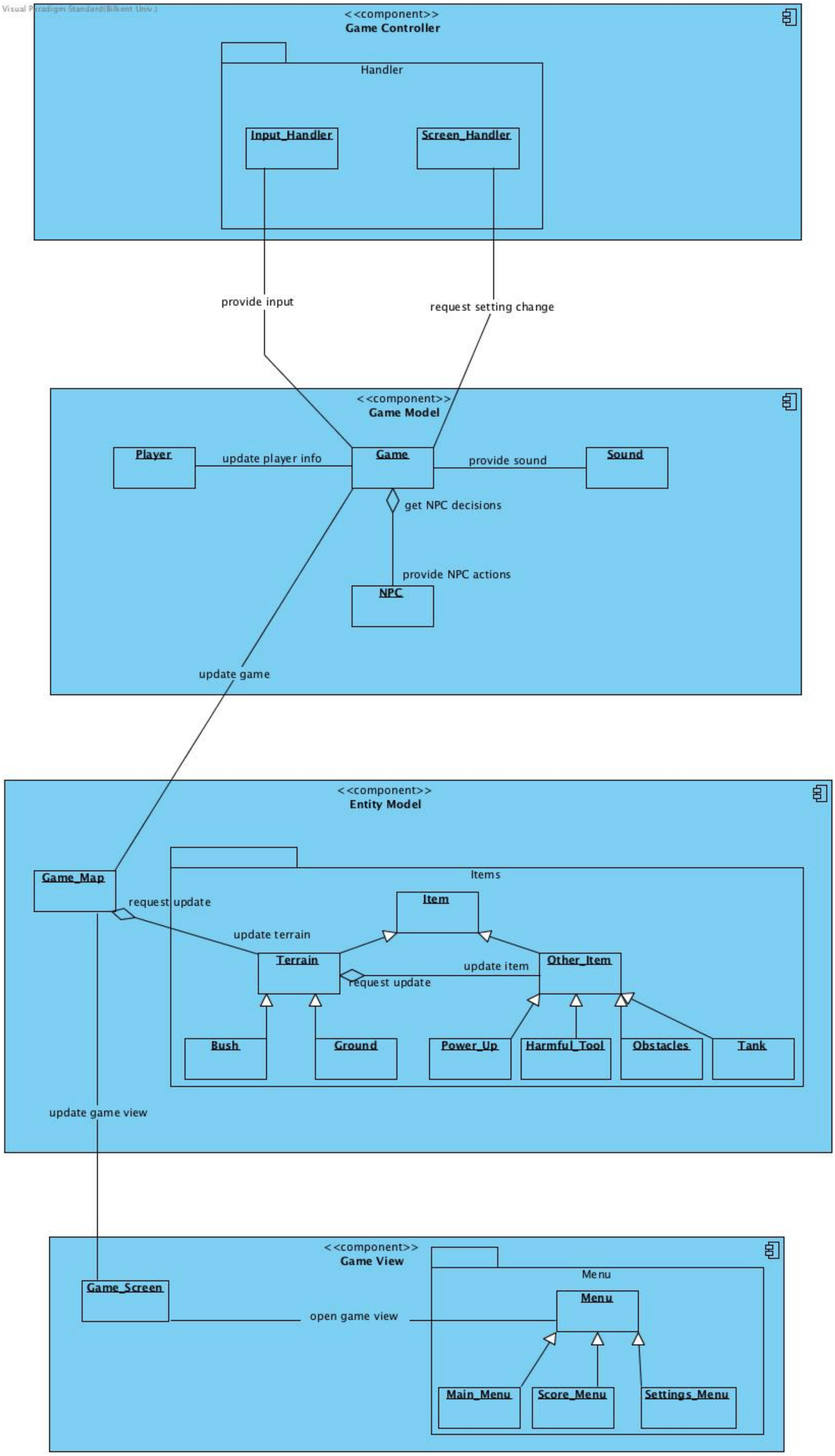

Visual Paradigm Standard(Bilkent Univ.)
<<component>>
Game Controller
Handler
Input_Handler
Screen_Handler
provide input
request setting change
<<component>>
Game Model
Player
update player info
Game
provide sound
Sound
get NPC decisions
provide NPC actions
NPC
update game
<<component>>
Entity Model
Game_Map
Items
request update
Item
update terrain
Terrain
update item
Other_Item
request update
Bush
Ground
Power_Up
Harmful_Tool
Obstacles
Tank
update game view
<<component>>
Game View
Menu
Game_Screen
Menu
open game view
Main_Menu
Score_Menu
Settings_Menu

Figure-2 The detailed subsystem decomposition

The first diagram in the above figures shows the subsystem decomposition of the software application by using the MVC. On the other hand, the second diagram in the above figure shows the subsystem decomposition in details with the links between subsystems and packages. The system is divided to four subsystems including a controller subsystem, a view subsystem and two model subsystems. Also, the subsystem decomposition has a closed architecture with four layers.

The controller subsystem consists of handler classes and depends on the model subsystems. It controls the flow of the inputs and the setting changes. Whenever a new setting is applied or input is sent from user, the controller subsystem request changes in the game logic to update the current game conditions.

The game model subsystem consists of game, player, sound and non-player character classes. It controls the game logic and make the decisions needed to carry forward the game. Game class takes the input and setting information from the controller subsystem and non-player character actions from the NPC class in order to decide to the flow of the game events. Game class updates the player information and requests sound whenever it is necessary in the game. The game model subsystem request changes from the entity subsystem in accordance with the game logic used by game class.

The entity model subsystem consists of game map class and the item classes inside the items package. It stores the objects and data relevant to the software application. Throughout the game, entity model subsystem makes updates to the entity objects of the game according to the requests of the game model subsystem. Game map updates the terrains inside the map and terrains updates the objects inside that particular terrain. Entity model class notifies the game view subsystem when any change occurs in the game map.

The game view subsystem consists of game screen class and menu classes inside the menu package. It displays the views on the screen and updates them when any notification from the game view subsystem is received. Game screen is responsible from the update of the game view due to the notifications of game map class. The menu package is responsible from the shifts between different menu views. Also, menu classes initialize the game screen in order to start the game.

2.3.Architectural Style

The model view controller architectural style is used for the system decomposition of the project. The choice of the architectural style aimed to decrease the dependencies between different subsystems and to increase the association between classes inside a subsystem. Therefore, two different model classes are designed in which game model interacts with the controller and controls the game logic and entity model serve as a repository for objects and notifies the view model when any change occurs in the stored data of the objects. Hence, a four-layered architecture is preferred for the decomposition of the subsystems with a closed architecture that the layers can only access the layer below them. Therefore, each layer in the subsystem decomposition uses the services of the layer below, which provides a more consistent system. The roles and functionalities of the objects and classes are considered to identify the subsystems of the software application.

2.4.Hardware/Software Mapping

The game will be implemented in Java programming language. Therefore, the computer system should support Java and have Java virtual machine on it. Also, the game will use keyboard inputs for the game control and mouse inputs for the shifts between menus. Hence, the necessary hardware requirements only include the keyboard and mouse. The scores and maps of the game will be kept in text files and sound effects will have .wav and .mp3 extensions. Therefore, the computer system should have support for the .wav, .mp3 and .txt files. The graphical requirements for the game consist of a computer system that can support java swing library objects since the swing library will be used for the development of the user interface.

2.5.Data Management

The sound files for the sound effects and text files for the maps and scores will keep the persistent data for our project. The text files will have the textual definitions of the maps as well as the high scores and these textual data will be kept after the execution of the program. Therefore, if necessary, the persistent objects of the project will be stored in the textual format after the termination of the entity objects in the game. Also, applied settings of the game will be kept in the text files as persistent objects so that the new games can start with the user specified settings. In case of system crash or data corruption, the retrieval of the scores, maps and settings might not be possible. However, since the maps and scores are generated before and after the game, the loss of the data will not affect the data flow in the game.

2.6.Access Control and Security

The user will not be able to edit or change any persistent object inside the text files from the outside of the game. Otherwise, the system will not have any restrictions or control over the user’s accesses to the relevant objects of the game in accordance with the rules of the game. Therefore, there is no need for the complicated authorization, authentication or security processes for this software program.

2.7.Boundary Conditions

Configuration: When the game is initialized, the game will load the default settings for the keyboard input and sound enabling if the user did not specified otherwise. If the user changes the settings of the game, the new settings will be stored in the text files after the termination of the game. The random generated maps will be created before the game starts and will be written to a text file. The game will be able to read the text file before the initialization of the game to generate the specified map in the text file.

Start-up and Shutdown: The game will come with an executable .jar file. The user can initialize the program by clicking the executable file. The program will be terminated when the user quits the game by clicking quit button. The active entity objects that are used inside the game and are not stored as persistent objects will be terminated after the game is finished.

Exception Handling: The game does not have any database or network connection that can result in exceptions. However, if there is an exception in the loading of the persistent objects or the initialization of the game and entity objects, the system will display an error message that shows the exception and continue the game by ignoring the exception if it is possible.

3. Subsystem Services

3.1.Design Patterns

The Swing library of Java will be used as user interface toolkit in our project. Since the swing library uses the composite design pattern to address the issues related with specialization of the user interface components and organization of the layout, the composite design pattern will be used in this project in order to address issues of the user interface design. Therefore, swing library will enable our project to organize the user interface components into hierarchy of classes by using the composite design pattern.

In addition to composite design pattern, Factory design pattern may be used to deal with the creation of the objects. This design pattern enables us to separate the game logic from the creation of the instance of the classes by using java interfaces. Therefore, factory method enables us to use a common interface for the creation of the objects and hide the instantiation of the object from the client.

3.2.Subsystem Interfaces

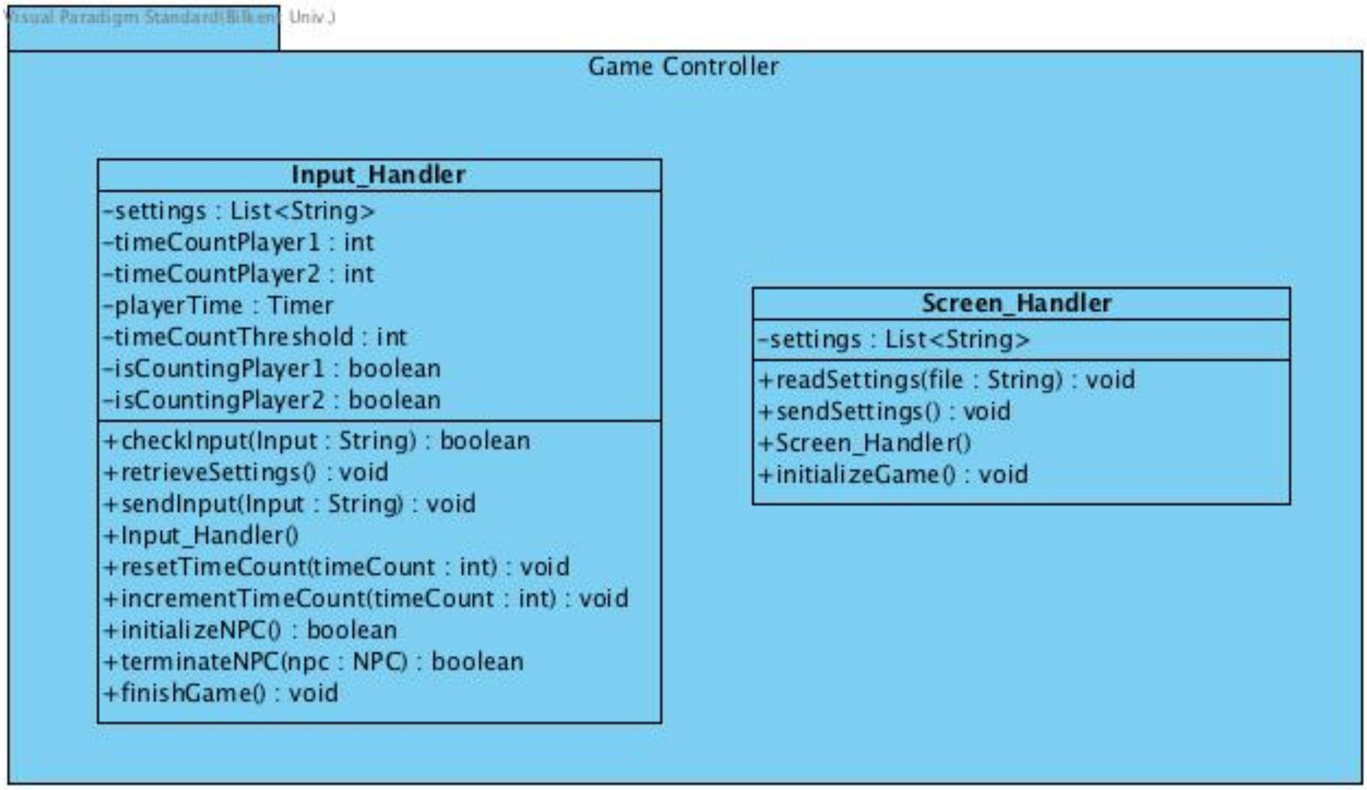
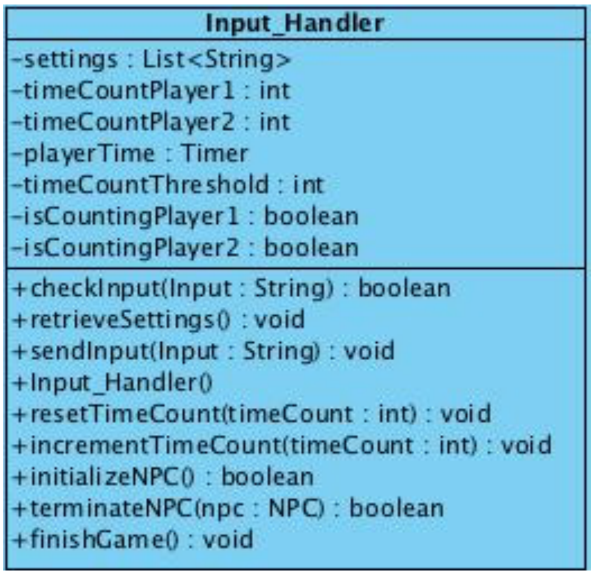

Visual Paradigm Standard(Bilkent Univ.)
Game Controller
Input_Handler
-settings : List<String>
-timeCountPlayer1 : int
-timeCountPlayer2 : int
-playerTime : Timer
-timeCountThreshold : int
-isCountingPlayer1 : boolean
-isCountingPlayer2 : boolean
+checkInput(Input : String) : boolean
+retrieveSettings() : void
+sendInput(Input : String) : void
+Input_Handler()
+resetTimeCount(timeCount : int) : void
+incrementTimeCount(timeCount : int) : void
+initializeNPC() : boolean
+terminateNPC(npc : NPC) : boolean
+finishGame() : void
Screen_Handler
-settings : List<String>
+readSettings(file : String) : void
+sendSettings() : void
+Screen_Handler()
+initializeGame() : void

Figure- 3 Game Controller Subsystem

The game controller subsystem is responsible for the control flow of the game. It controls the initialization and termination of the AI as well as the flow of the input in the game. It also checks the validity of the input and decides whether the game continue as a single player or multiplayer game. It handles the updates of the setting options in the game and controls the initialization of the game model classes. It interacts with the game model subsystem to manage the control flow of the inputs and the settings of the game. It also interacts with the event listeners in order to obtain the necessary inputs from the user

Input Handler Class



Input_Handler

-settings : List<String>
-timeCountPlayer1 : int
-timeCountPlayer2 : int
-playerTime : Timer
-timeCountThreshold : int
-isCountingPlayer1 : boolean
-isCountingPlayer2 : boolean

+checkInput(Input : String) : boolean
+retrieveSettings() : void
+sendInput(Input : String) : void
+Input_Handler()
+resetTimeCount(timeCount : int) : void
+incrementTimeCount(timeCount : int) : void
+initializeNPC() : boolean
+terminateNPC(npc : NPC) : boolean
+finishGame() : void

Attributes

settings: This attribute contains the list of the settings as string values. Each string in the list corresponds to a keyboard key name.

time count player 1, time count player 2: These attributes keep the count of the time that passed after the last input of the player 1-2.

is counting player 1, is counting player 2: These attributes determine whether the player 1-2 is in the game. If the user has left the game and AI initialized, the corresponding attribute for that player takes false. Otherwise, it takes true.

player time: This attribute is the instance of the timer class that generates events for every 100 millisecond.

time count threshold: This attribute determines the threshold for deciding whether the player left the game after some amount of time or not.

Methods

check input: This method checks the validity of the user input by comparing it with the current settings of the game by using the settings attribute. It takes the input of the player as a string from the event listener classes. If the input is valid, it returns true. If the valid input is coming from a player that has left the game before, this method calls the terminate npc method in order to return back to a multiplayer game.

retrieve settings: This method gets the current settings from the screen handler class and updates the settings attribute at the beginning of each game.

send input: This method sends the valid inputs to the game class. It takes the input of the player as a string from the event listener classes.

input handler: This constructor initializes the instance of the input handler class at the initialization of the game.

reset time count: This method reset the time count of a particular player if there is an valid input from that player.

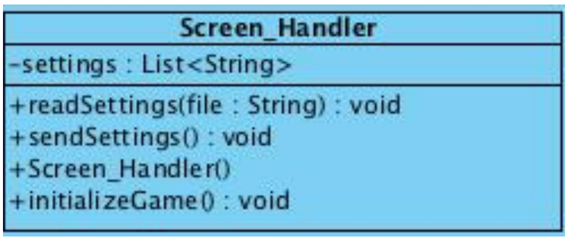
increment time count: This method increments the time count of a particular player in each time an event is generated by the player time timer. Also, it checks whether the new time count value is greater than the time count threshold.

initialize npc: This method initializes an AI for a single player game when the time count for a particular player exceeds the time count threshold. After the initialization of the AI, the time count for that player remains zero.

terminate npc: This method terminates an AI to start a multiplayer game when it is called by the check input method. After the termination of the AI, this method restarts the time count for the player that joined the game.

finish game: If is counting player 1 and is counting player 2 take false, this method finishes the game by calling the appropriate methods from the game class since there is no remaining player in the game.

Screen Handler Class



```
Screen_Handler
-settings : List<String>
+readSettings(file : String) : void
+sendSettings() : void
+Screen_Handler()
+initializeGame() : void
```

Attributes

settings: This attribute contains the list of the settings as string values. Each string correspond to a keyboard key name.

Methods

read settings: This method reads the current settings of the game from the text file and updates the settings list.

send settings: This method sends the settings to the game class to implement the new settings to the game after the settings are read from the text file

initialize game: This method calls the game class to initialize a new game when the game screen is created.

screen handler: This constructor initializes the instance of the screen handler class at the initialization of the game and calls the read settings method for the first time.

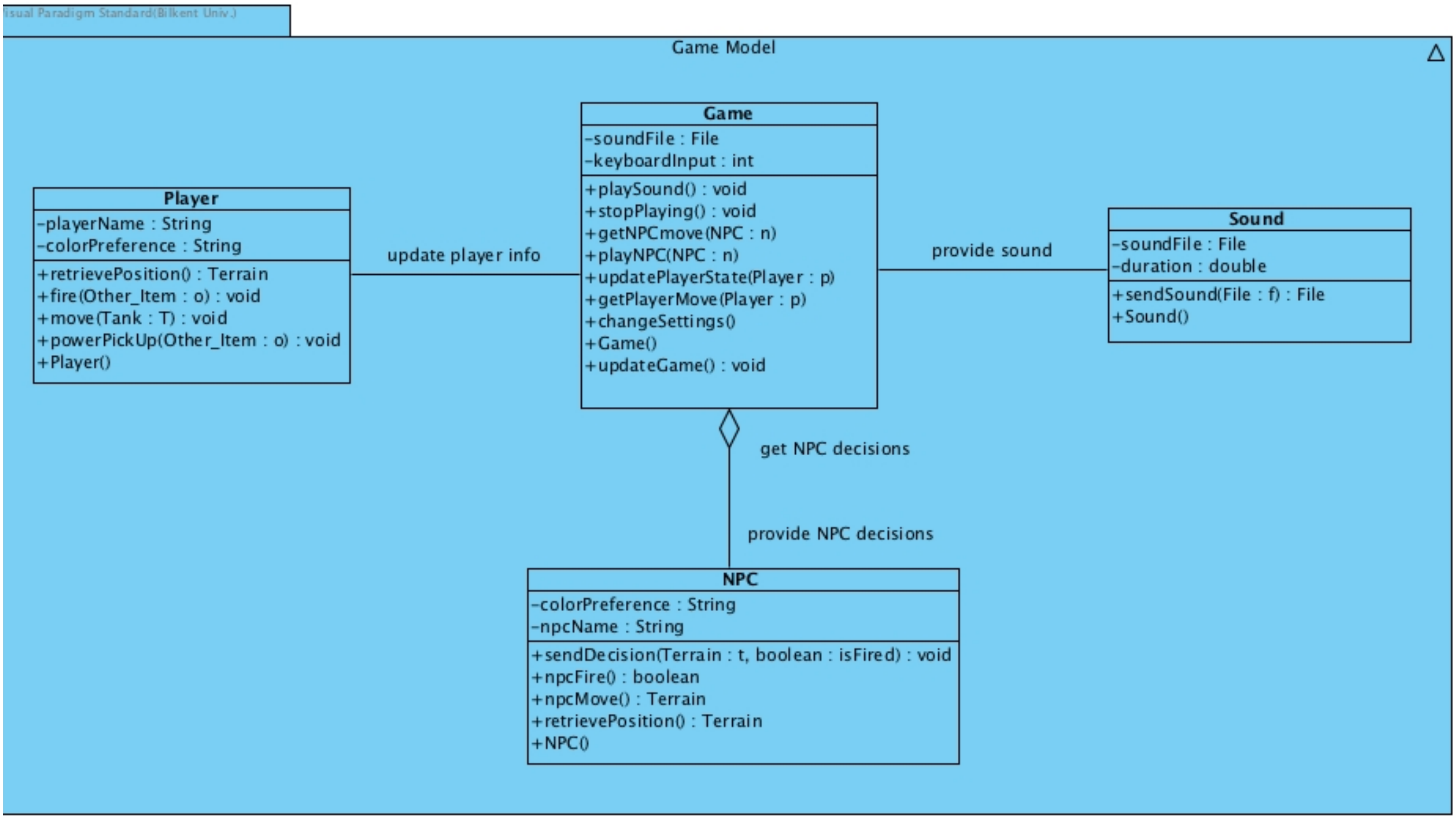
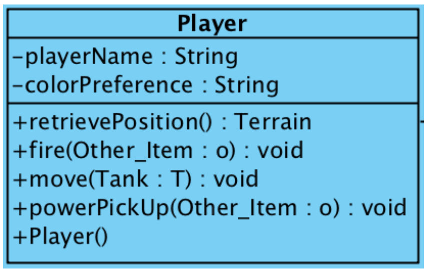


Figure-4: Game Model Subsystem

The game model subsystem is responsible for playing and stopping the actions sounds, deciding on NPC players' moves and playing for them, handle players' move and handle the key inputs from the keyboard. The model interacts with Game Controller models' classes (Input Handler & Screen Handler) to manage the movements and sends the updates to the Game Map class of Entity Model. Inside the model, Player class creates the Player objects that players will own during the game. NPC class will create an NPC instance that will make its own decisions. Lastly, Sound class creates a sound and sends it to the Game class for it to be played.

Player Class


Player
-playerName : String
-colorPreference : String
+retrievePosition() : Terrain
+fire(Other_Item : o) : void
+move(Tank : T) : void
+powerPickUp(Other_Item : o) : void
+Player()

Attributes:

player name: This attribute defines player's names as a String at the beginning of every new game.

color preference: Players can choose between two colors and this attribute serves the purpose of distinguishing between two players.

Methods:

retrieve position: This method returns the Terrain that the player stands at the time of method call.

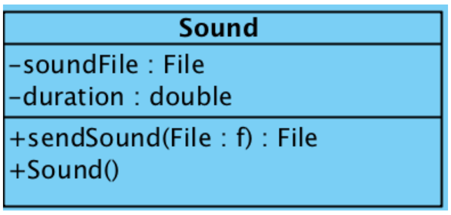
fire: Fire method commands the Tank to fire a bullet or a mine on its call.

move: This method sends a request to the Game class to move itself.

power pick up: This method is called when a power-up is picked up and power-up's effects update either the Tank's or game's status.

Player: Constructor for Player class where its attributes are initialized and an object instance of the class is created.

Sound Class



Attributes:

sound file: The .wav/.mp3 file to be played on actions in the game. A unique sound file defines it.

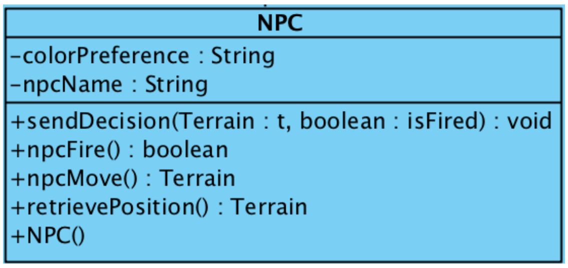
duration: Sound file's duration that will be represented in double.

Methods:

sendSound: This attribute will send the Sound objects' details to the Game class for it to be handled inside the game.

Sound: Constructor for Sound class where its attributes are initialized and an object instance of the class is created.

NPC Class



```
NPC
-colorPreference : String
-npcName : String
+sendDecision(Terrain : t, boolean : isFired) : void
+npcFire() : boolean
+npcMove() : Terrain
+retrievePosition() : Terrain
+NPC()
```

Attributes:

color preference: NPC tanks will always be set to gray throughout the gameplay.

npc name: This attribute defines NPC's names as a String at the beginning of every new game. NPC's names will be picked from a predefined name list randomly.

Methods:

send decision: Send decision method's parameters is fired (Boolean) and t (Terrain) are the values to be passed to Game class. Game class sends a request to the Game Map class to update the screen accordingly.

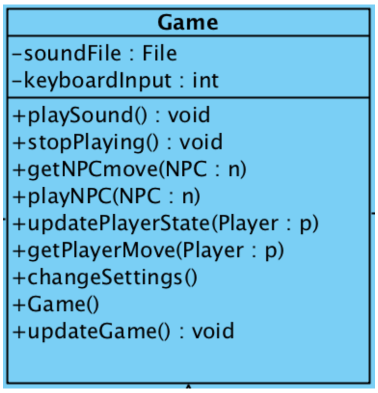
npc fire: Fire method commands the Tank to fire a bullet or a mine on its call. There will be no input to this method because a player will not have the control of it. This method return a Boolean and this value will be given to send decision method to make a move appropriately.

npc move: This method returns the desired Terrain to be moved. The return value will be passed to send decision method.

retrieve position: This method returns the Terrain that the player stands at the time of method call.

NPC: Constructor for NPC class where attributes are initialized and an object instance of the class is created.

Game Class



Attributes:

sound file: The .wav/.mp3 file to be played on actions in the game. A unique sound file defines it.

keyboard input: The integer key value of the input that is obtained from the keyboard from the player. The request

Methods:

play sound: This method start playing the sound file with the duration of that Sound object has.

stop playing: This method stops any ongoing playing sounds in the game. If there are no sound playing, no action is taken by the method.

get npc move: Having the NPC move sent from the NPC class, this method gets the move that the NPC instance has sent.

play npc: Play NPC method takes the necessary actions that were requested by an NPC object and sends the update.

get player move: This method gets the moves that were made by the players and saves them to be used for further updates.

update player state: Having obtained the movement information from the get player move method, player’s state (health, Terrain, firing state) is updated and sent to the Game Map class’s instance.

change settings: This method handles the setting change request that comes from the Screen Handler, Game Controller Model.

update game: The request method that interacts with the Game Map class to make an update to the map.

Game: Constructor for Game class where attributes are initialized and an object instance of the class is created.

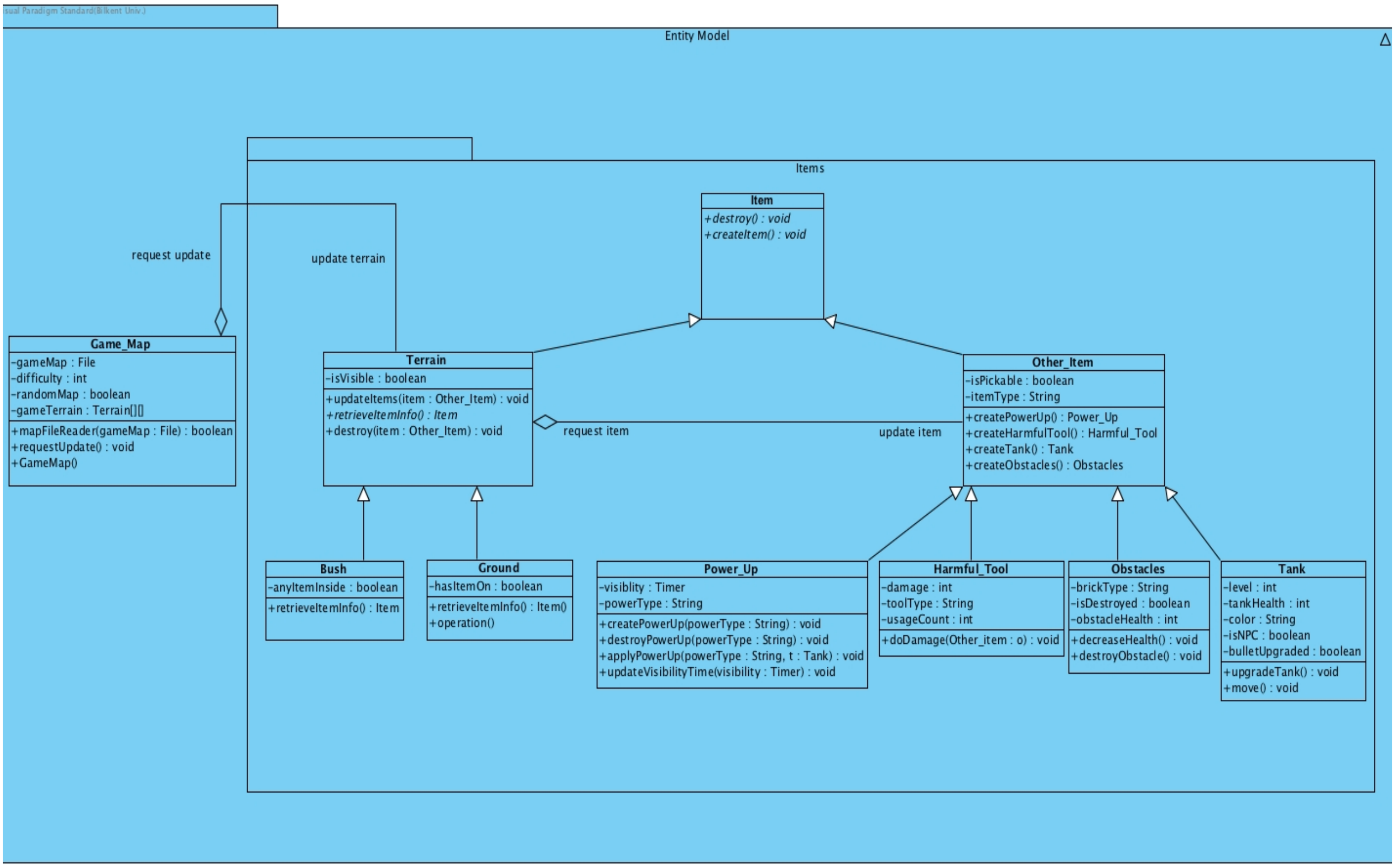
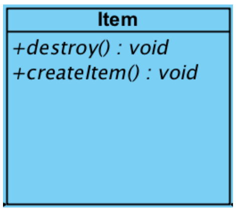


Figure-5: Entity Model Subsystem

The entity model subsystem is responsible for the data management in the game and the updates on the map items. The model contains an Items package, which defines the objects (with their interactions) in the game. In addition to Items package, Game Map class interacts with the Game class from Game Model to update the game's state with respective update requests to the map. Preset maps are defined with a .txt file and when setting up the game, the map is converted to the object model with Terrain objects, which can contain Item objects.

Item Class



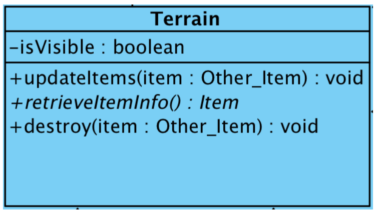
Attributes: This class has no attributes because the classes that extend this class have their own attributes that are unique to them

Methods:

destroy: This method is an abstract method for the classes that will extend this class during the creation of respective objects. The method aims to destroy the instance of that object.

create item: This method is an abstract method for the classes that will extend this class during the creation of respective objects. The method aims to create an instance of the respective class' object

Terrain Class


Terrain
–isVisible : boolean
+updateItems(item : Other_Item) : void
+retrieveItemInfo() : Item
+destroy(item : Other_Item) : void

This class serves the purpose of having a container for the items that can exist on the map.

Attributes:

isVisible: This attribute defines an item's visibility through either bush or ground. The tank is not visible under a bush whereas it's visible on the ground. Is visible is used in the manner of deciding item's visibility in the map.

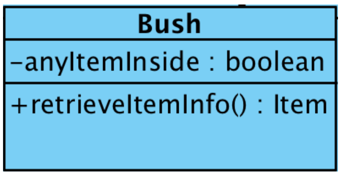
Methods:

update items: This method organizes terrains' state and update the Terrain object to place an item on it to update and inform the Game Map instance with the change.

retrieve item info: This method provides the needed information of the object residing in the terrain. By knowing this information, updating or destroying terrains with Item objects is made easier.

destroy: This method overrides the destroy method that is provided in the Item class as abstract. The method destroys the Item that is on that Terrain so that the game can flow and the map can be updated.

Bush Class



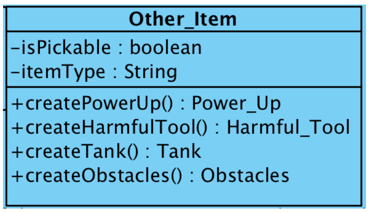
Attributes:

any item inside: Since bushes in the game make the object inside itself invisible, this attribute is a Boolean to distinguish whether there exists an object inside or not. Updating operations can be done with having the information of the object that is inside the bush.

Methods:

retrieve item info: This method provides the needed information of the object residing in the Terrain object. By knowing this information, updating or destroying terrains with Item objects is made easier for the Terrain object.

Other_Item Class


Other_Item
-isPickable : boolean
-itemType : String
+createPowerUp() : Power_Up
+createHarmfulTool() : Harmful_Tool
+createTank() : Tank
+createObstacles() : Obstacles

This class contains the objects that are not stacked to a terrain throughout the game (e.g. bushes)

Attributes:

is pickable: This attribute defines whether an item is pickable from the terrain or not. If it’s pickable then that means it is a power up.

item type: This attribute is a String that can be either one of the following: “Power up”, “Harmful”, “Tank”, “Obstacle”. By having this attribute, methods that create respective objects are called.

Methods:

create power up: This method creates a Power Up object that is visible on the map if it’s not under a bush. A Power Up object is created and returned after the call of the method.

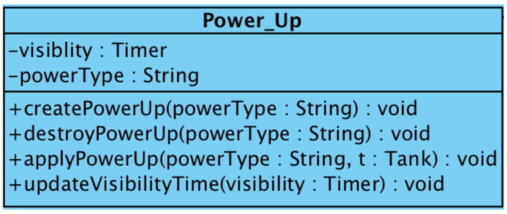
create harmful tool: This method creates a Harmful Tool object that can belong to any tank in the game. Harmful Tools are either mines (up to 5) or tank bullets (infinitely many and can be upgraded). A Harmful Tool object is created and returned after the call of the method.

create tank: This method creates a Tank object that is visible on the map if it’s not under a bush. A Tank object is created and returned after the call of the method.

create obstacles: This method creates an Obstacles object that can either be an Ordinary Brick or a Steel Brick. The object is created and placed to the respective Terrain object. An Obstacles object is created and returned after the call of the method.

The map and the respective Terrain objects are updated after every creation of an object.

Power_Up Class


Power_Up
-visiblity : Timer
-powerType : String
+createPowerUp(powerType : String) : void
+destroyPowerUp(powerType : String) : void
+applyPowerUp(powerType : String, t : Tank) : void
+updateVisibilityTime(visibility : Timer) : void

Attributes:

is pickable: This attribute defines whether an item is pickable from the terrain or not. If it's pickable then that means it is a power up.

item type: This attribute is a String that can be either one of the following: "Power up", "Harmful", "Tank", "Obstacle". By having this attribute, methods that create respective objects are called.

Methods:

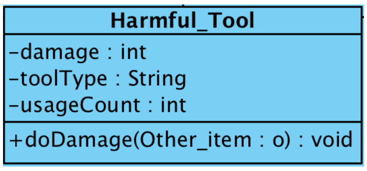
create power up: This method creates a Power Up object that is visible on the map if it's not under a bush. A Power Up object is created and returned after the call of the method.

create harmful tool: This method creates a Harmful Tool object that can belong to any tank in the game. Harmful Tools are either mines (up to 5) or tank bullets (infinitely many and can be upgraded). A Harmful Tool object is created and returned after the call of the method.

create tank: This method creates a Tank object that is visible on the map if it's not under a bush. A Tank object is created and returned after the call of the method.

create obstacles: This method creates an Obstacles object that can either be an Ordinary Brick or a Steel Brick. The object is created and placed to the respective Terrain object. An Obstacles object is created and returned after the call of the method.

Harmful_Tool Class



Attributes:

damage: Damage attribute of Harmful Tool object defines the damage that is being caused by utilizing the object. The health of the target object will be decreased as the value of damage attribute on a successful hit.

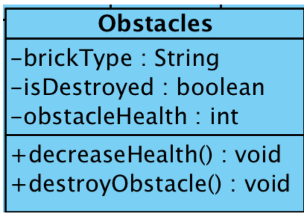
tool type: Type of a Harmful Tool object is defined by a String attribute. They can either be mines or bullets. It must be specified during the creation of an object.

usage count: A tank can have infinite number of bullets therefore usage count attribute is set to be the maximum integer for a bullet so that it a tank won't run out of bullets. For mines, usage count starts off with 5 and decreases every time it is used.

Methods:

do damage: This method takes Other Item object as a parameter and Other Item object is used to do damage on the target object. Usage count attribute of the Harmful Tool object is decreased on each do damage method call.

Obstacles Class



Attributes:

brick type: Since there are two types of obstacles in the game - steel brick and ordinary brick - they are defined as a String attribute. Obstacles' type is defined with this attribute.

is destroyed: Obstacles objects can be destroyed by Tank objects' Harmful Tools' do damage method. Thus they can be destroyed. In order to keep track of what obstacles are destroyed in the game, this Boolean attribute is defined.

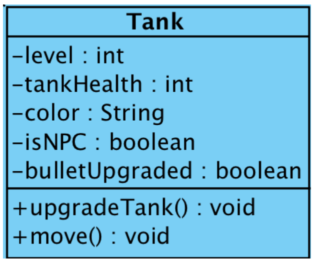
obstacle health: Obstacles objects can be damaged by Tank objects' Harmful Tools. Obstacles objects are kept with obstacle health attribute to check whether it encountered enough amount of damage to get destroyed. (0 health)

Methods:

decrease health: If an Obstacles object gets damaged by a Harmful Tool object, this method is called to decrease the obstacle's health. The health decrease depends on the Harmful Tool's damage attribute.

destroy obstacle: Obstacles with zero health are destroyed with this method to update the respective Terrain object's view in the game.

Tank Class



Attributes:

level: Not being dependent on tanks' controller (is it being controlled by a player with a keyboard or a NPC) tanks can be upgraded up to fourth level with power-ups. This integer attribute specifies tanks' levels. Tank levels affect tanks' health.

tank health: Since this is a multiplayer game, players can damage each other and tanks can get a power-up to get more health. This attribute keeps track of tanks' health in the game.

color: Tank objects controlled by human players can be either yellow or green. NPC tanks are gray by default. Color attribute defines Tank objects' colors.

is NPC: This Boolean attribute defines how the tank is being controlled. If it is set true, the computer with its own logic will control the tank.

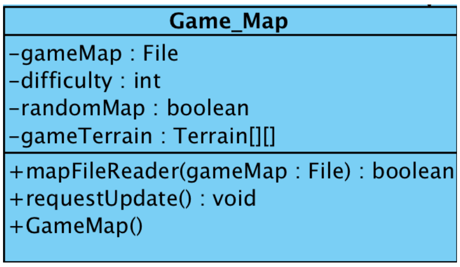
bullet upgraded: If the Harmful Tool object of a Tank's bullet is upgraded, this attributed is set to true. Setting this attribute will result in respective tanks' bullet to damage more.

Methods:

upgrade tank: By picking up the tank upgrade power-up, the Tank object will be upgraded. This method will be called upon upgrade.

move: Move method will be called when tank is moving. This will update Game Map to update the Terrain objects.

Game_Map Class



Attributes:

game map: This attribute is a text file where the map is defined with obstacles' and power-ups unique initials. Maps can be created and loaded with the information that is provided with this attribute.

difficulty: The game has 3 preset difficulties and they are set with an integer attribute of the Game Map class.

random map: An optional random map can be generated by the players. The map is not guaranteed to have the best in-game performance in terms of obstacles. This boolean attribute determines the map type.

game terrain: This attribute is a 2D Terrain object array that builds up the whole map. With map file reader method, game map attribute is read and results in game terrain attribute with the items that were specified in the file.

Methods:

map file reader: This method reads the map file (game map attribute) to build the map with Terrain objects. The map is properly formed via this method.

request update: Game Map handles the changes on the map with requests to the Terrain objects. Terrain object is passed to request update to change the respective Terrain.

game map: This is the constructor of the Game Map class that initializes the instances of it and calls the terrain updates to form the map with the objects placed.

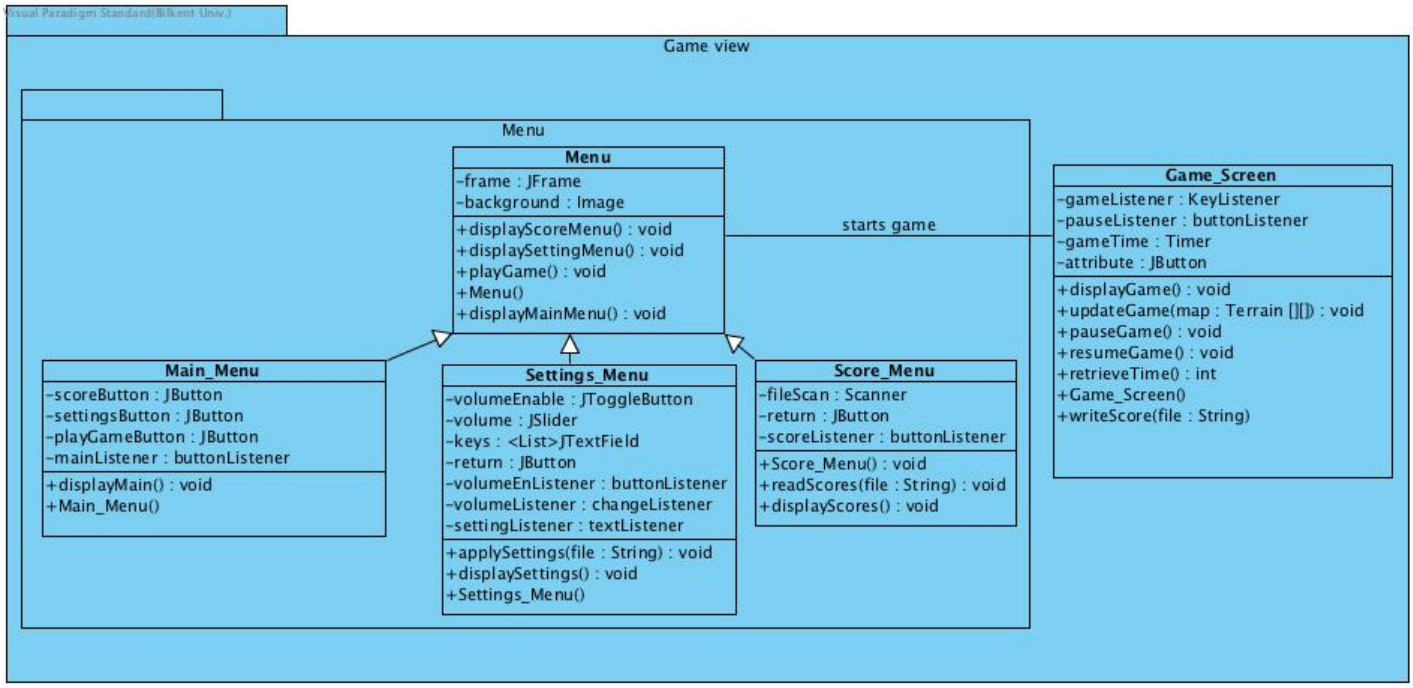
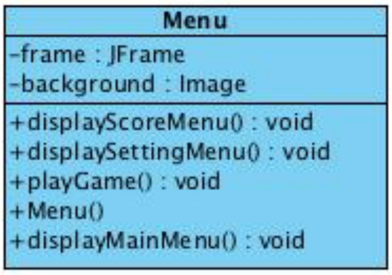

Visual Paradigm Standard(Bilkent Univ.)
Game view
Menu
Menu
-frame : JFrame
-background : Image
+displayScoreMenu() : void
+displaySettingMenu() : void
+playGame() : void
+Menu()
+displayMainMenu() : void
starts game
Game_Screen
-gameListener : KeyListener
-pauseListener : buttonListener
-gameTime : Timer
-attribute : JButton
+displayGame() : void
+updateGame(map : Terrain [][]) : void
+pauseGame() : void
+resumeGame() : void
+retrieveTime() : int
+Game_Screen()
+writeScore(file : String)
Main_Menu
-scoreButton : JButton
-settingsButton : JButton
-playGameButton : JButton
-mainListener : buttonListener
+displayMain() : void
+Main_Menu()
Settings_Menu
-volumeEnable : JToggleButton
-volume : JSlider
-keys : <List>JTextField
-return : JButton
-volumeEnListener : buttonListener
-volumeListener : changeListener
-settingListener : textListener
+applySettings(file : String) : void
+displaySettings() : void
+Settings_Menu()
Score_Menu
-fileScan : Scanner
-return : JButton
-scoreListener : buttonListener
+Score_Menu() : void
+readScores(file : String) : void
+displayScores() : void

Figure-6: Game View Subsystem For The User Interface

The game view subsystem is responsible for the graphical components of the game. It contains the menu package, which draws the menu screens and displays the settings and scores information. Also, it contains the game screen that displays the graphical components of the game. The menu classes initialize the game screen at the beginning of the game.

Menu Class


Menu
-frame : JFrame
-background : Image
+displayScoreMenu() : void
+displaySettingMenu() : void
+playGame() : void
+Menu()
+displayMainMenu() : void

Attributes

frame: The instance of the JFrame class which is used as the main frame of the game.

background: This attribute is the image that is displayed at the background of the frame.

Methods

display score menu: This method gets the score menu panel to the screen and removes the other panels from the screen.

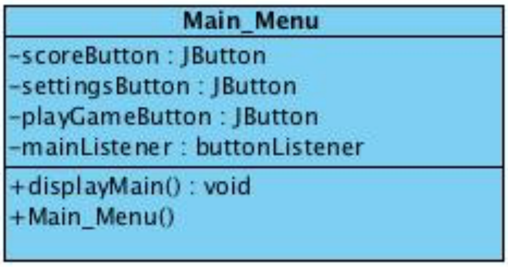
display setting menu play game: This method gets the settings menu panel to the screen and removes the other panels from the screen.

display main menu: This method gets the main menu panel to the screen and removes the other panels from the screen.

play game: This method initializes the game screen by creating the game screen panel and removing all other panels from the screen. Also, it creates the screen handler and calls the initialize game method to create the new game with the current settings.

menu: This is the constructor of the menu class that initializes the instances of it and calls the display main menu for the first time to display the graphics.

Main Menu Class


Main_Menu
-scoreButton : JButton
-settingsButton : JButton
-playGameButton : JButton
-mainListener : buttonListener
+displayMain() : void
+Main_Menu()

Attributes

score button, settings button and play game button: These buttons are responsible for getting the events that triggers the creation of other panels.

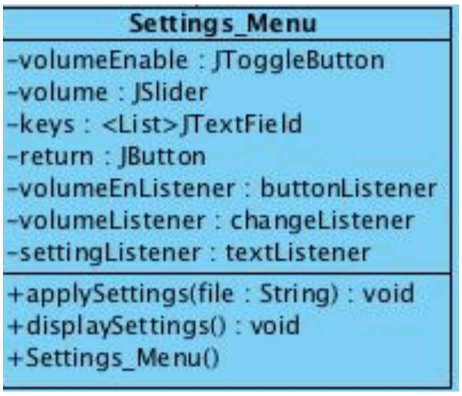
main listener: This listener listens to the buttons that are specified above and calls the appropriate methods when the events occur.

Methods

display main: This method calls the paint components to draw the graphics to the screen if necessary. Also, it is responsible for the layout and design of the panel.

main menu: This is the constructor of the main menu class that initializes the instances of it. The main menu is the first panel that is initialized in the program.

Settings Menu



```
Settings_Menu
-volumeEnable : JToggleButton
-volume : JSlider
-keys : <List>JTextField
-return : JButton
-volumeEnListener : buttonListener
-volumeListener : changeListener
-settingListener : textListener
+applySettings(file : String) : void
+displaySettings() : void
+Settings_Menu()
```

Attributes

volume Enable: This attribute is the instance of the toggle button class of the java swing library. It is responsible for the enabling and disabling of the sound effects.

volume: This attribute is the instance of the slider class of the java swing library. It is responsible for the control of the volume density of the sound effects.

keys: This attribute is the list of the text fields for the keys that control the player input in the game. Each text field gets one specification of key for one action of the player.

return: This attribute is the instance of the button class of the java swing library. It is responsible for returning back to the main menu panel.

setting Listener, volume listener and volume enable listener: These listeners listen to the events generated by the above instances of the java swing library classes to get the appropriate changes to the settings made by user.

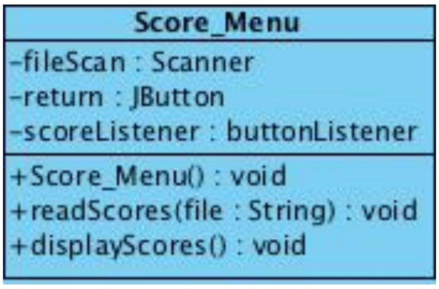
Methods

apply settings: This method writes the new settings of the game to a text file according to the events caught by the listeners.

display settings: This method displays the current settings on the screen and draws the graphics to the screen if necessary.

settings menu: This is the constructor of the settings menu class that initializes the instances of it and calls the display settings to display the graphics.

Score Menu



Score_Menu

-fileScan : Scanner
-return : JButton
-scoreListener : buttonListener

+Score_Menu() : void
+readScores(file : String) : void
+displayScores() : void

Attributes

file Scan: The instance of the Scanner class which is used to scan the text file that contains the high scores to display them.

return: This attribute is the instance of the button class of the java swing library. It is responsible for returning back to the main menu panel.

score Listener: This listener listens to the button events of the return button.

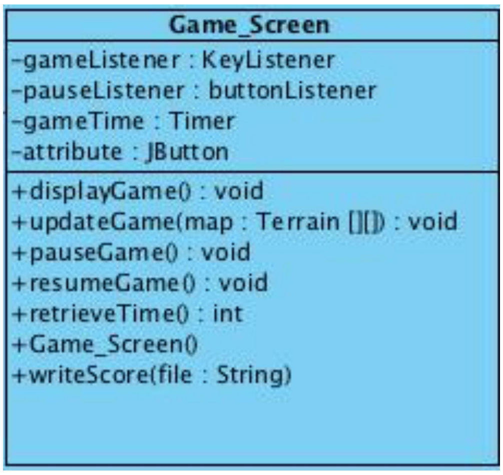
Methods

read scores: This method reads the highest scores of the game from the text file using the scanner instance.

display scores: This method displays the scores and draws the graphics to the screen.

score menu: This is the constructor of the score menu class that initializes the instances of it and calls the display scores to display the graphics and scores on the screen.

Game Screen



```
Game_Screen
-gameListener : KeyListener
-pauseListener : buttonListener
-gameTime : Timer
-attribute : JButton
+displayGame() : void
+updateGame(map : Terrain [][]) : void
+pauseGame() : void
+resumeGame() : void
+retrieveTime() : int
+Game_Screen()
+writeScore(file : String)
```

Attributes

game listener: This listener listens to the keyboard inputs of the user and sends the inputs to the input handler class.

pause: This attribute is the instance of java button class of the swing library which is used to stop and resume the game.

pause listener: This listener listens to the pause button specified above.

game time: This attribute is the instance of the timer class that counts down from a constant time to zero until the game is over.

Methods

display game: This method displays the initial components of the game when it is initialized. It also displays the map panel that the user selects the generated map for the game.

update game: This method is responsible for the updating of the graphics on the screen when a change occurs in the game. It gets the description of the game map that will be displayed as an input.

pause game and resume game: This methods are responsible for the pause and resume operations that brings the settings panel to the screen when the pause listener catches an event.

retrieve time: This method returns the time information obtained from the game time as an integer value.

game screen: This is the constructor of the game screen class that initializes the instances of it and calls the display game for the first time to display the graphics and components on the screen.

write score: This method writes the scores to the text file at the end of the game if the score is one of the highest scores.