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 objects, animations very smoothy. 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 extension will be used

so the sounds of the game don't affect the game's performance since the uncompressed .wav files do not need to be decompressed, which reduces the effect on the 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 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

NPC: Non-player character

1.4. References

Bruegge, Bernd and Allen Dutoit. Object-Oriented Software Engineering Using UML, Patterns, and Java. 3rd ed. Edinburgh: Pearson, 2014. Print.

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

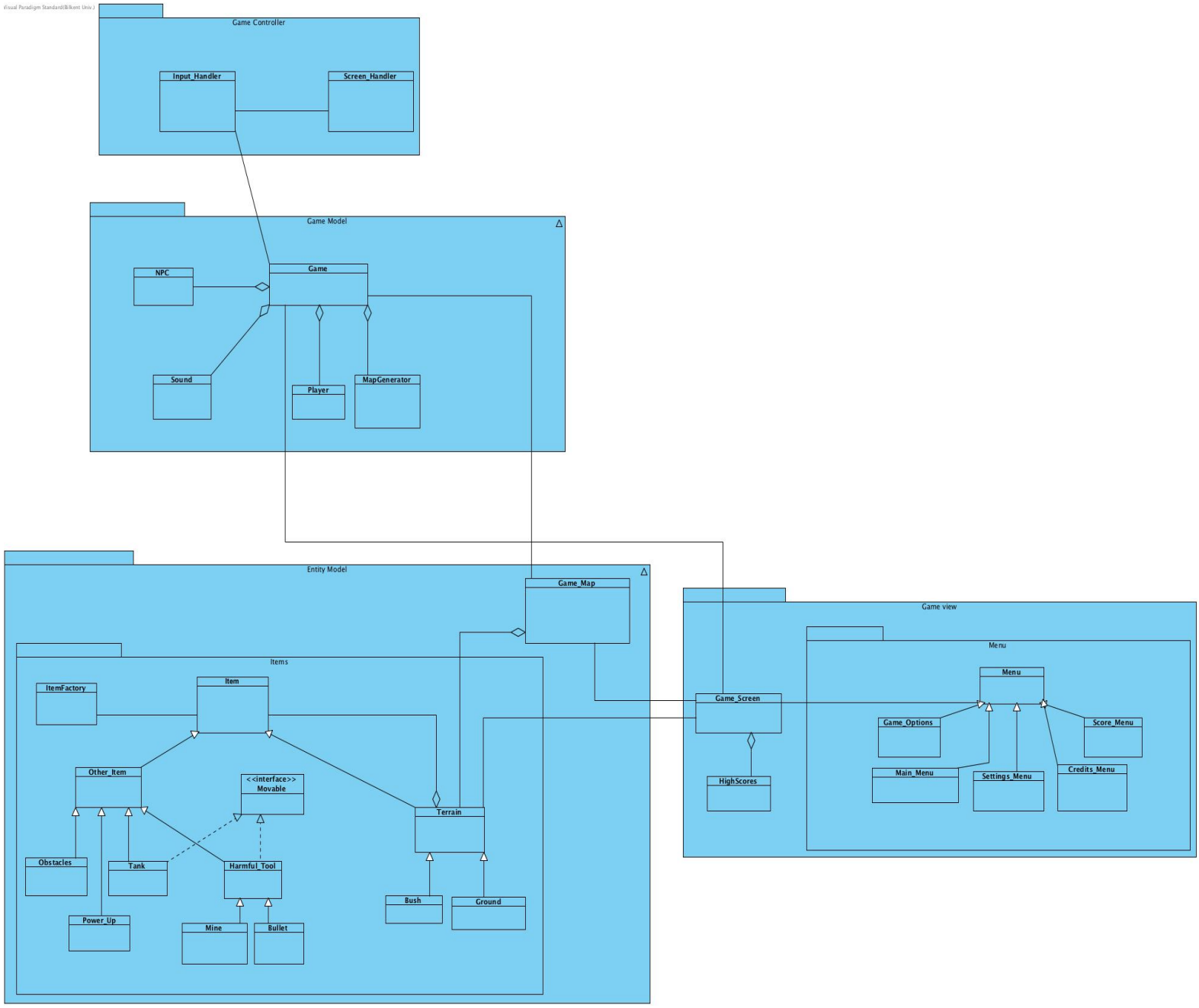


Figure-1 Subsystem Decomposition with Dependencies

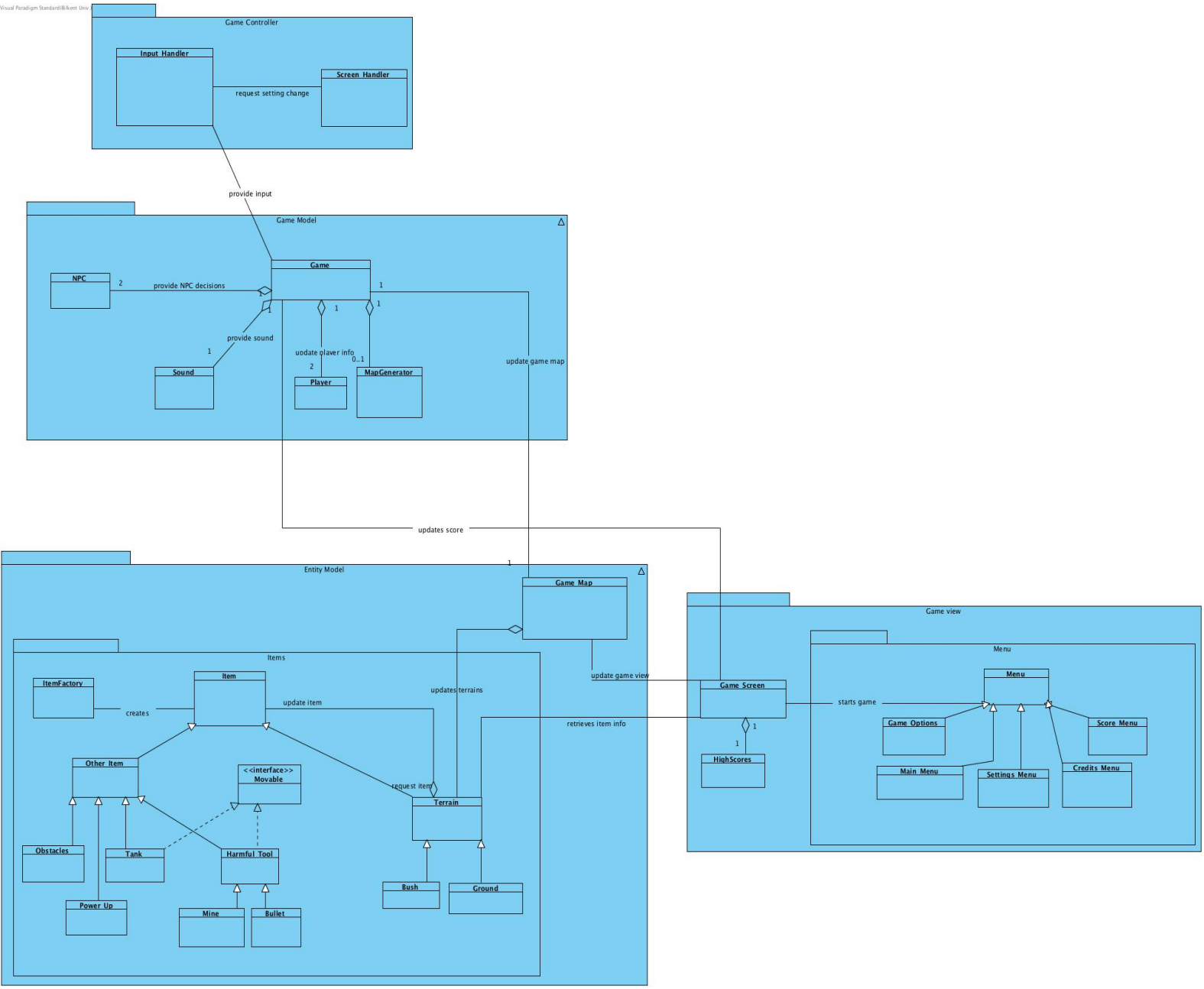


Figure-2 The detailed subsystem decomposition

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 consist 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 consist 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 consist 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, text files for the maps of the game, which could be predefined or randomly generated, and scores will be kept as the persistent data for our project. The text files will have the textual definitions of the maps as well as the highscores 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

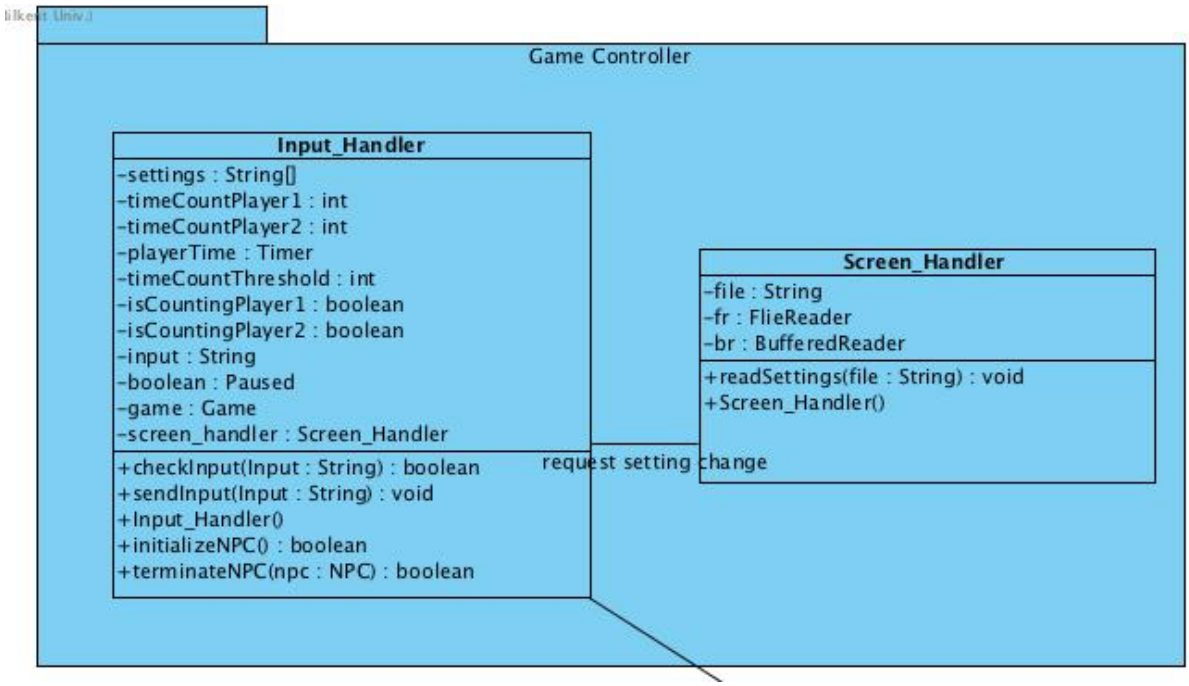
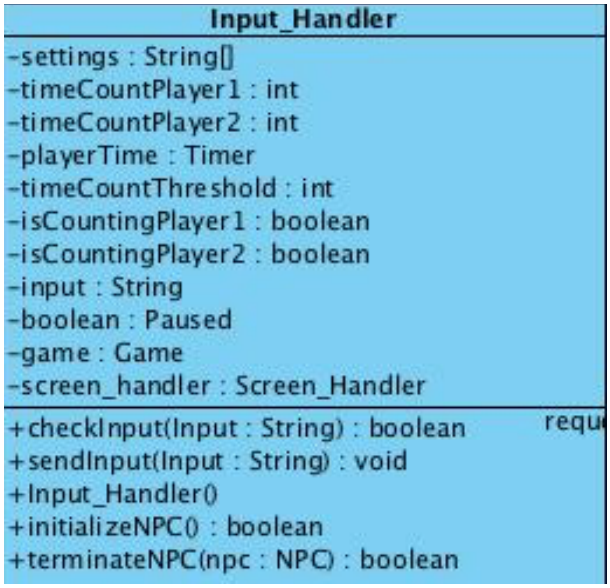


Figure- 3 Game Controller Subsystem

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



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.

input: This attribute keeps track of the entered inputs from users. The value of this input changes how tanks operate and does the appropriate changes on screen handler.

paused: This boolean attribute of the Input Handler class checks whether the game was paused or not for future reference to the game. If the timer is greater than zero but the game is stopped, then the game is paused.

game: An instance of the game logic that enables the input handler class to operate on.

screen_handler: An instance of the Screen_Handler class to pass the necessary changes on the inputs to the shown game screen accordingly.

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.

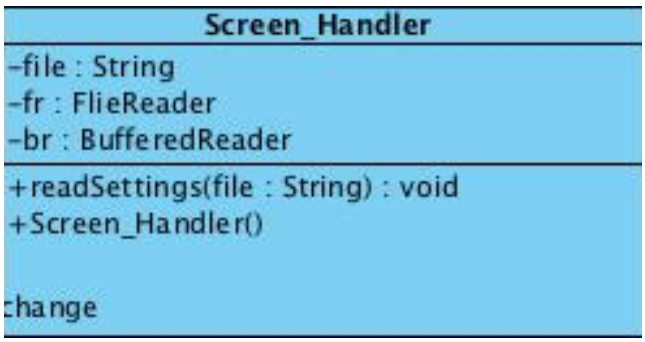
send input: This method sends the valid inputs to the game class. It takes the input of the player as a string from the event listener classes.

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

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.

Screen Handler Class



Attributes

file: This attribute contains the list of the settings as a single line in a txt file. Each token in the file corresponds to a keyboard key name.

fr, br: These attributes reads the inputs from the settings file.

Methods

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

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.

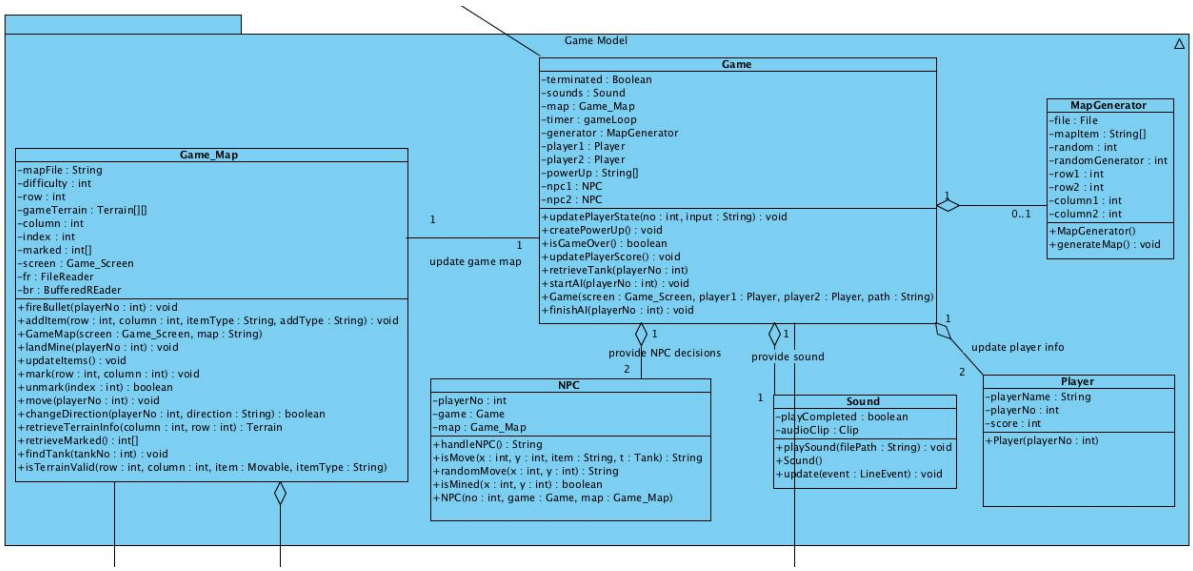
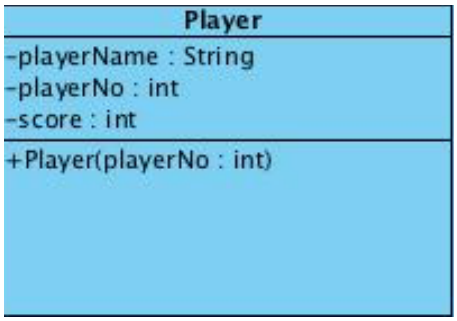


Figure-4: Game Model Subsystem

The game model subsystem is responsible for playing and stopping the actions and sounds, deciding on NPC players' moves and playing for them, handling players' move and handling 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. The getter and setter methods for the variables will be used to retrieve and update the variables.

Player Class



Attributes:

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

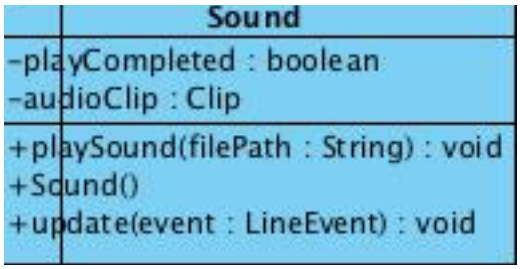
player no: This attribute defines player's number, either 1 or 2 since the game is played with 2 players at most.

score: This integer attribute of Player class keeps track of player instances' scores in the game.

Methods:

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

Sound Class



Attributes:

play completed: The boolean attribute of the Sound class underlining the clip's playing status. False if still playing.

audio clip: The .wav/.mp3 file to be played on actions in the game. It is defined by a unique sound file.

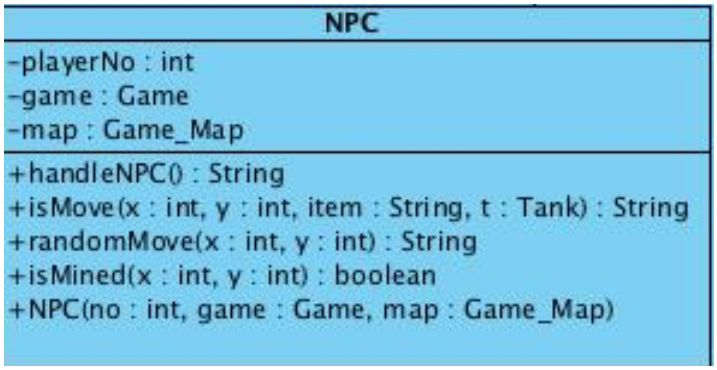
Methods:

playSound: This attribute will send the Sound objects' details to the Game class for it to be handled inside the game. With this attribute, a specific sound will be played for each action taken in the game.

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

update: Updating sound's playing status with this method to let other classes know what will be the next sound to be played.

NPC Class



Attributes:

player no: This attribute defines player's number, either 1 or 2 since the game is played with 2 players at most.

game: An instance of the game logic that enables NPCs to operate on.

game map: An instance of the game map so that NPCs move and actions can be reflected to the active game map.

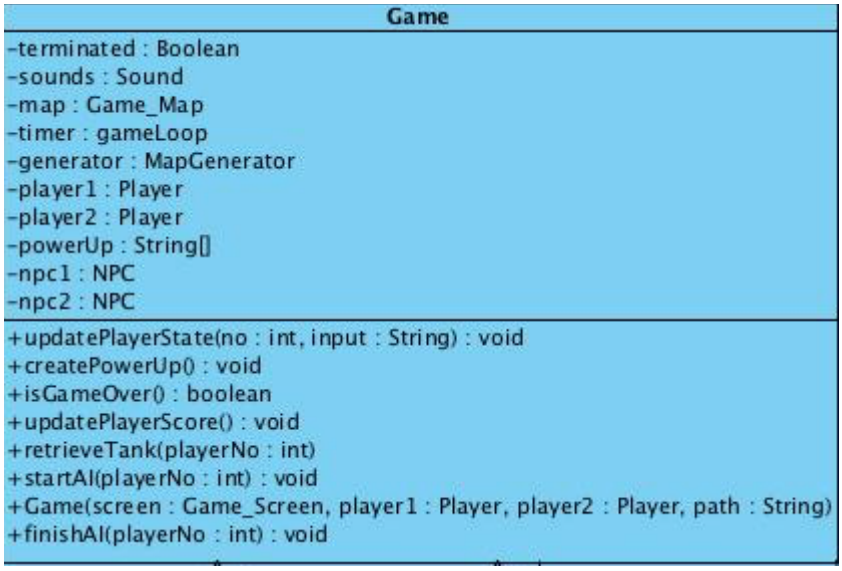
Methods:

handle npc: Handles npcs' decisions by sending the respective values to the Game class. Game class sends a request to the Game_Map class to update the screen accordingly.

is move, random move, is mined: The methods that enables the NPC instances in the game to move on their own decisions. These methods have coordinate inputs as parameters and String or boolean return types depending on methods' intentions.

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

Game Class



Attributes:

terminated: A boolean attribute keeping game's state if it was terminated.

sounds: The .wav/.mp3 file to be played on actions in the game. It is defined by a unique sound file.

map: An instance of Game_Map instance to update and reflect the changes to the game map that were made during game's flow.

game loop: The main clock running to operate objects in the game. This timer has the role to finish the game if the remaining time hits zero.

generator: An attribute that has a resulting value from the generated map.

player1, player2, npc1, npc2: Players in the game, some of them can be null if NPC takes the control after 5 seconds of timeout.

power up: A string array of the power-ups in the game.

Methods:

create power up: With this method, a random power-up is generated in the game for tanks to pick up.

start ai, finish ai: start ai and finish ai method takes the necessary actions for NPC's turns. When an input is not given from the user in 5 seconds, start ai method is called. finish ai method is called when an input is read but the input handler and player takes the control.

retrieve tank: This method gets the tank for a given player number as a parameter.

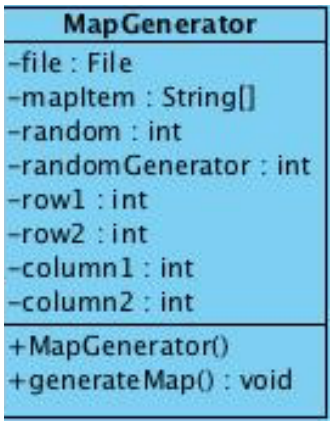
update player state: Having obtained the movement information from the get player move method, player's state (health, Terrain, firing state) is updated and sent to the Game_Map class's instance.

update player score: Given a player number to the method, the players' score is incremented by the action player took.

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

is game over: Getter method for game's state in order to provide access to game's state between other classes.

Map Generator Class



Attributes:

file: The txt file where the random map is going to be generated.

map item: A string array of all available items in the game, indexed from zero to access the items.

random, random generator: RandomGenerator attribute generates a random integer and the integer is stored in the attribute random. This number directly chooses a map item from the mapItem String array.

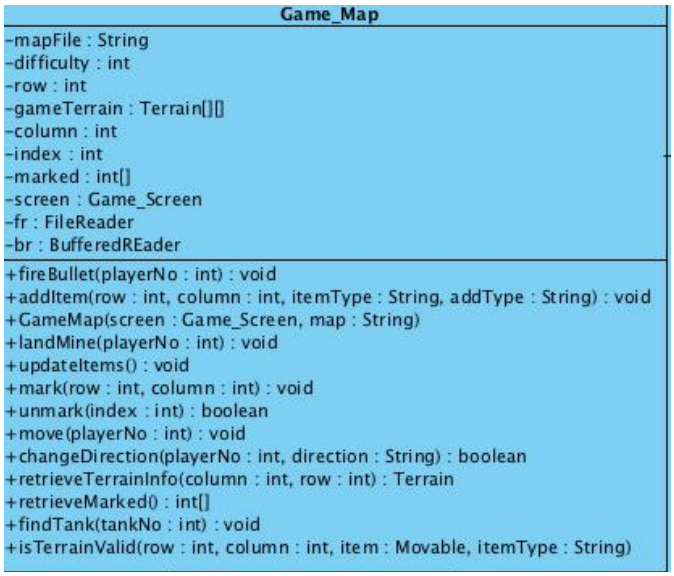
row1, row2, column1, column2: Integer attributes to specify the position of the randomly selected map item with the random generator.

Methods

generate map: This method creates a totally random map with the available terrain items.

map generator: Constructor for Map Generator class where its attributes are initialized and an object instance of the class is created.

Game Map Class



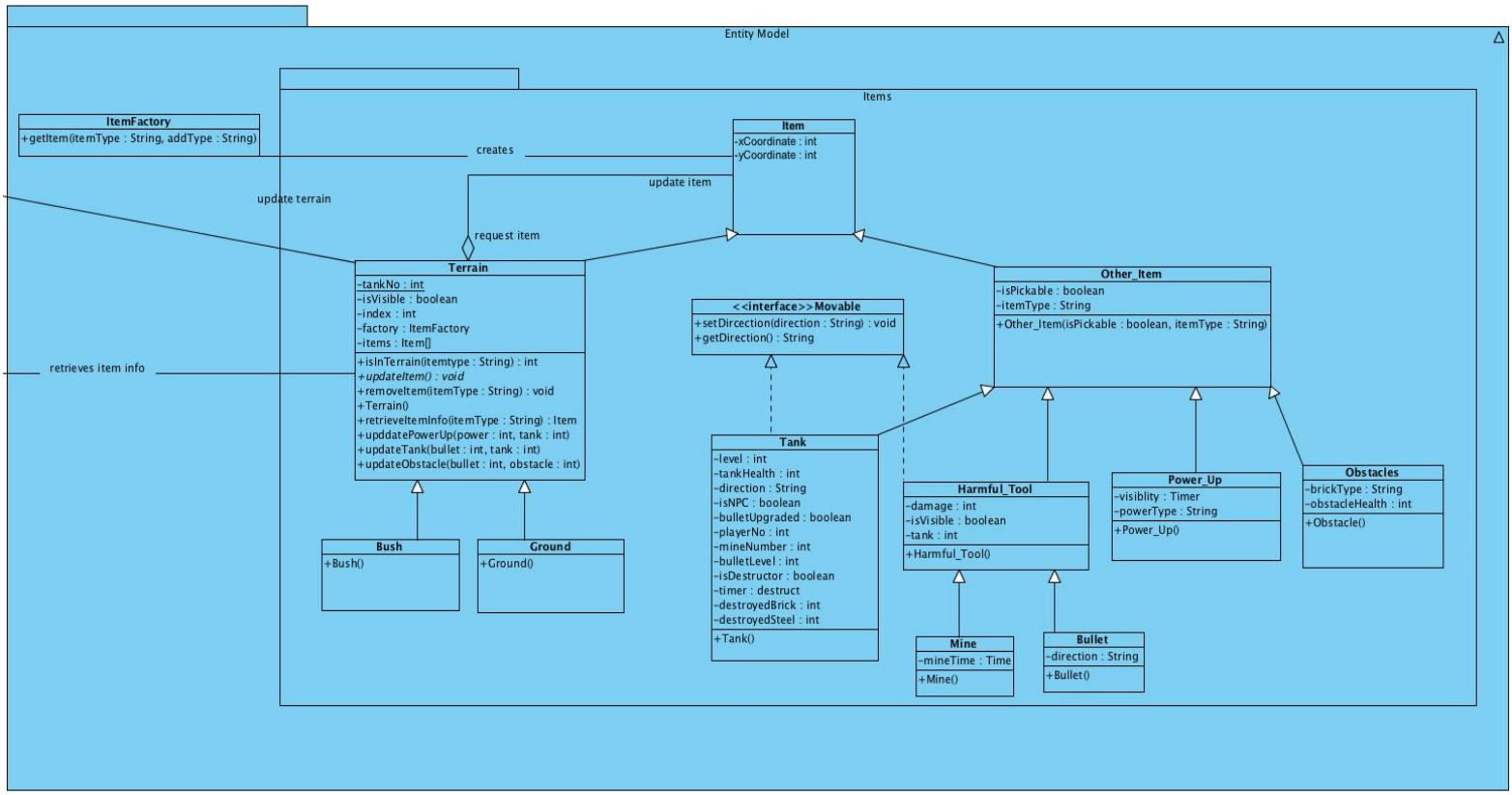
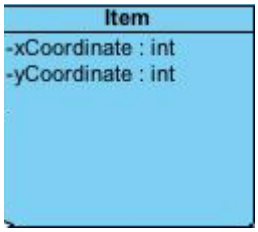


Figure-5: Entity Model Subsystem

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.

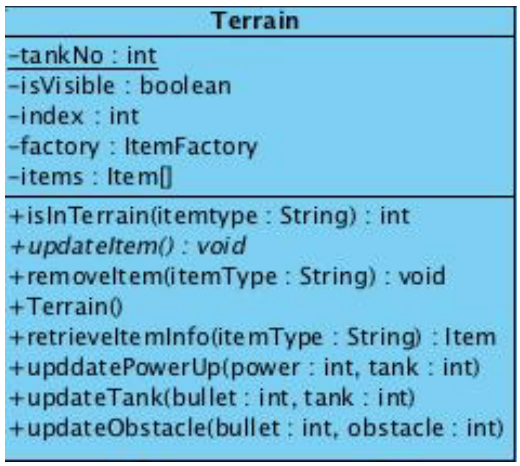
Item Class



Attributes:

x coordinate, y coordinate: Specifies the item instance's position on the 10x10 grid of the game map.

Terrain Class



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

Attributes:

is visible: 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. isVisible is used in the manner of deciding item's visibility in the map.

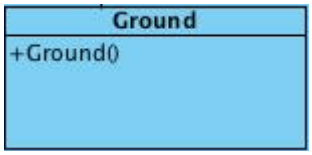
Methods:

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.

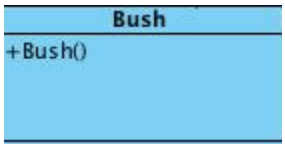
Ground Class



Methods:

Ground: Constructor for Ground class where an object instance of the class is created.

Bush Class

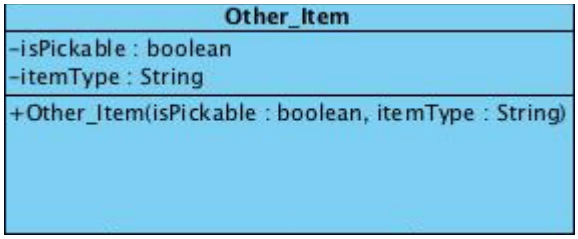


Bushes in the game make the object inside itself invisible.

Methods:

Bush: Constructor for Bush class where an object instance of the class is created.

Other_Item Class



This class contains the objects that are not sticked 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:

Other Item: Constructor for Other Item class where an object instance of the class is created. This constructor can create instances of powerups, harmful tools, tanks and obstacles depending on the parameters given.

Power_Up Class

Attributes:

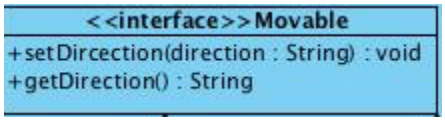
visibility: This attribute is a Timer which specifies the visibility time of the power up, namely it's life time.

power type: This attribute is a String that can be either one of the following: "Brick Rider", "Bullet Upgrade", "Mine Pick-up", "Tank Upgrade", "Extra Time". By having this attribute, methods can create powerups by their names.

Methods:

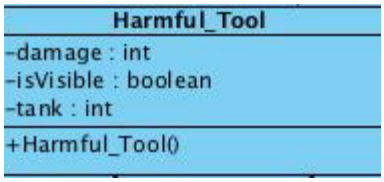
Power Up: Constructor for Power Up class where an object instance of the class is created.

Movable Interface



This interface defines the methods for bullets' and tanks' directions in the game.

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.

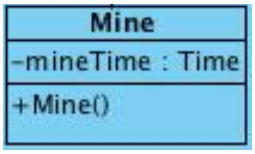
is visible: This attribute sets Harmful Tool's visibilty. Harmful Tool (Mines) can either be visible after usage and can be invisible after explosion. Bullet are visible as long as they don't hit anything.

tank: The Harmful Tool object's belongingness is defined with this integer attribute.

Methods:

Harmful Tool: Constructor for Harmful Tool class where an object instance of the class is created.

Mine Class



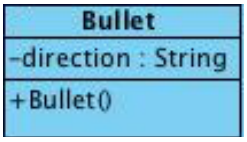
Attributes:

mine time: Mines visibility is set by a Time attribute in the class. It is a preset value.

Methods:

Mine: Constructor for Mine class where an object instance of the class is created.

Bullet Class



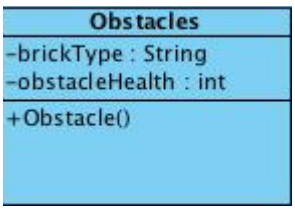
Attributes:

direction: Bullets in the game can either go "up", "down", "right" or "left". This attribute specifies its direction at a time point.

Methods:

Bullet: Constructor for Bullet class where an object instance of the class is created.

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.

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:

Obstacle: Constructor for Obstacle class where an object instance of the class is created.

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' healths.

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.

direction: Tanks in the game can either go “up”, “down”, “right” or “left”. This attribute specifies its direction at a time point.

is NPC: This boolean attribute defines how the tank is being controlled. If set true, the tank will be controlled by the computer with its own logic.

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.

player no: This attribute defines tanks’ player number, either 1 or 2 since the game is played with 2 players at most.

mine number: Tank’s total mine count.

bullet level: Attribute specifying Tank bullets’ level. It can only be upgraded with the Bullet Upgrade powerups.

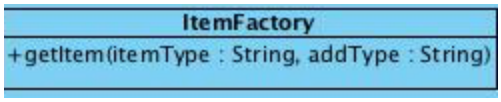
destroyed brick, destroyed steel: Integer attribute that keeps track of the total amount of destroyed ordinary and steel bricks for final score calculation.

destruct: A timer attribute for destruction of the tanks in game.

Methods:

Tank: Constructor for Tank class where an object instance of the class is created.

Item Factory Class



Methods:

get item: This method behaves as if it is a factory and creates an instance of the given String parameter of that specific object.

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.

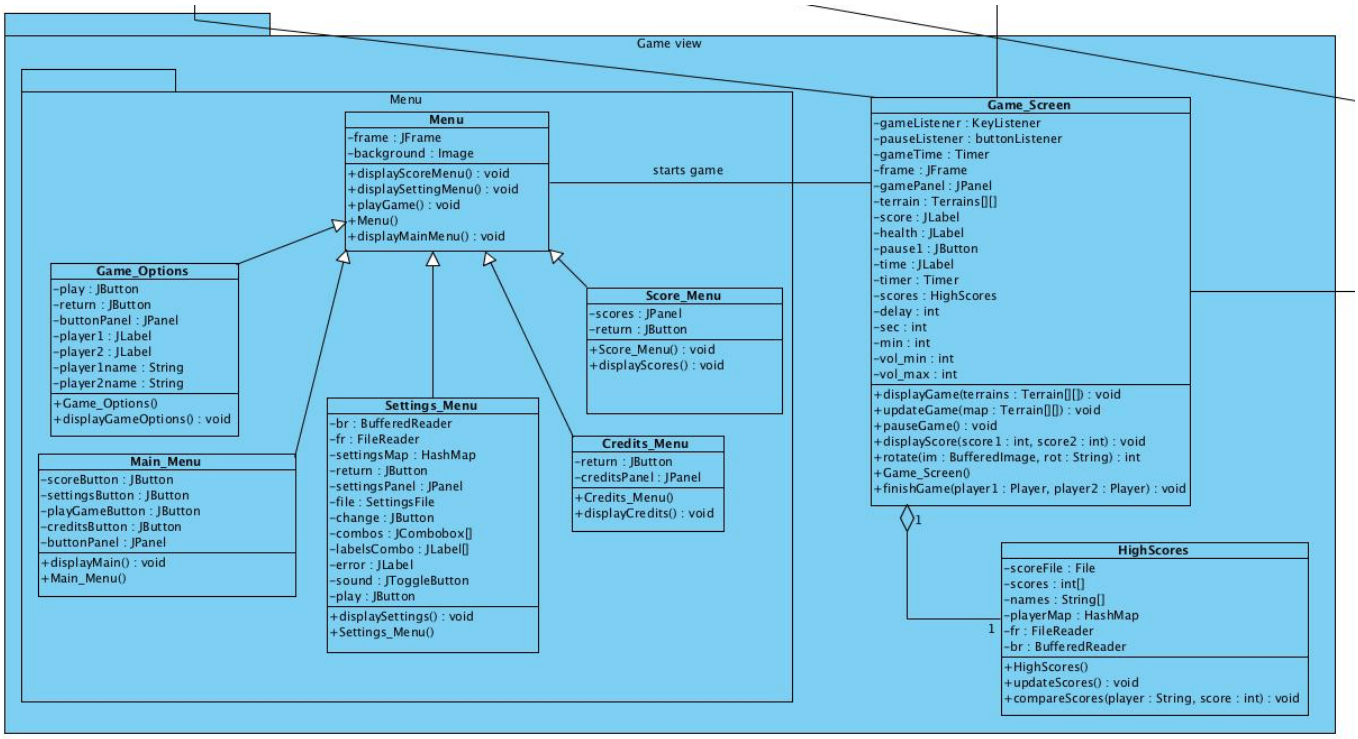
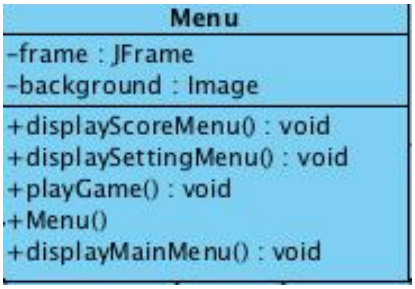


Figure-6: Game View Subsystem For The User Interface

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 game screen is initialized by the menu classes at the beginning of the game.

Menu Class



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.

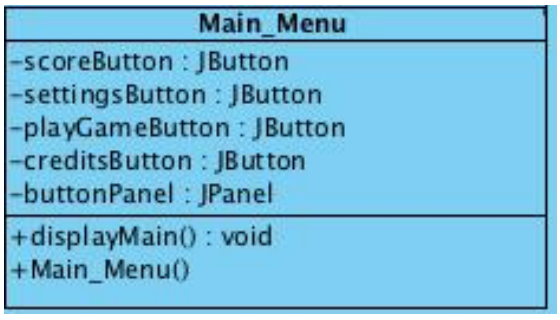
display setting menu play game: This method gets the settings menu panel to the screen and removes the other panels from the screen.

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



Attributes

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

button panel: The panel where all the buttons are added to and contains all of the buttons.

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

Attributes

fr, br: These attributes reads the inputs from the settings file.

play: This attribute is the instance of the button class of the java swing library. It is responsible for starting the game and initializing an instance of game screen.

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.

settings map: the default and selected settings key mapping is kept in this HashMap. These settings are written in the settings file later on.

settings file: The txt file where all the integers representations of the keys are kept with a space delimiter. All keys in the file must be unique.

settings panel: The panel where all the buttons, labels and comboboxes are added to .

change: The JButton instance that changes the chosen settings via the comboboxes.

sound: The toggle button to enable/disable game sound.

combos, labels combo: Combos is an array of JComboBox that shows users 40 different keys to select their controls from. The game control which the selection will affect is labeled with JLabel array labelsCombo, with same indexes on both arrays.

error: This attribute is the instance of the label class of the java swing library. It is responsible for showing errors when multiple keys are assigned to the same control in the game.

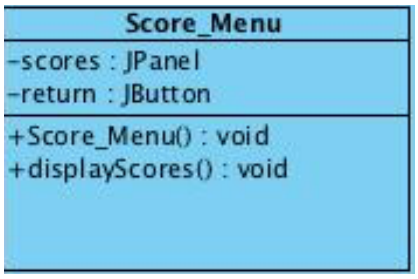
setting Listener, volume listener and volume enable listener: These listeners listen to the events generated by the above instances of the java swing library classes to get the appropriate changes to the settings made by user.

Methods

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



Attributes

scores: The panel where the button and scores textfield is added to and wraps them all.

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

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.

Credits_Menu Class

Attributes

credits panel: The panel where the button and scores textfield is added to and wraps them all.

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.

credits listener: This listener listens to the button events of the return button.

Methods

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

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

Game_Options Class

Attributes

button panel: The panel where all the buttons are added to and contains all of the buttons.

play: This attribute is the instance of the button class of the java swing library. It is responsible for starting the game and initializing an instance of game screen.

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.

player1, player2: Instances of the JLabel class of the java swing library. They contain the players' names.

player 1 name, player 2 name: The attribute which holds the names that were taken as inputs from the users.

credits listener: This listener listens to the button events of the return button.

Methods

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

Game Options: This is the constructor of the Game Options class that initializes the instances of it and calls the display game options to display the graphics and the options on the screen.

Game Screen Class

Attributes

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

pause 1: 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.

frame: The JFrame object where the whole panels, buttons, labels and most importantly the game lies in.

game panel: The panel where all the buttons are added to and contains all of the buttons.

terrain: A 2D array of Terrain objects to create and operate the game view for Game Screen.

scores: An attribute of Highscores object to update the highscores at the end of the game after players' scores are obtained.

Methods

display game: This method displays the initial components of the game when it is initialized. It also displays the mapPanel 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.

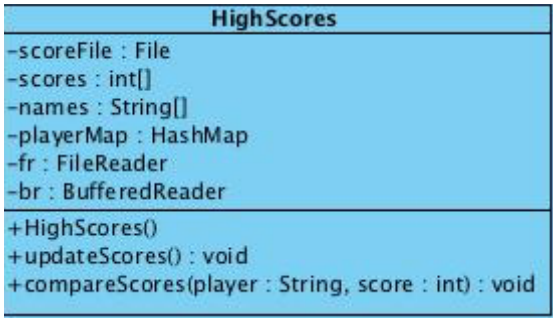
pause game and resume game: This methods are responsible for the pause and resume operations that brings the settings panel to the screen when the pause listener catches an event.

rotate: Since the image resources for this project have a single direction, they have to be rotated to 3 other directions for smooth graphics. This method in Game Screen class rotates the given input image.

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.

finish game: This method finishes the game for the Game Screen when the timer hits zero.

HighScores Class



Attributes:

score file: The txt file where all highscores are kept in the format "<string> <integer>"

scores: The integer array to do a comparison between scores so that new scores after an ending game can be easily added to the existing score file.

names: The attribute that keeps the names in the highscores and the input names at runtime. It is utilized in rewriting the file for updates scores.

fr, br: These attributes reads the inputs from the highcores file.

Methods:

Highscores: Constructor for Highscores class where an object instance of the class is created.

compare scores: This method compares a players score with the scores in the existing scores file.

update scores: This method makes a method call to compare scores method and updates the score file with the new scores included in their proper places.

3.3. UML Class Diagram