Design Report

CS 319-1

Group 1A, Team Java++

Project: Road Block

Members:

Denizhan Soydaş, 21502231

Deniz Ufuk Düzgün, 21402164

Kaan Atakan Öztürk, 21302164

Cavid Gayıblı, 21500636

Selim Can Gülsever, 21601033

Serkan Delil, 21501289

Contents Table

1. Introduction

1.1. Purpose of the System

Road Block is a game that we are planning to develop. The main objectives and game play of it are as follows: There is a thief in the game that is trying to escape from the police. The user's main objective is to block the paths which the thief may use in order to get away. The game is played on a map that offers 36(6x6) squares. This map offers various spaces in which the user can place differently shaped blocks and the puzzle in it is to place all the blocks where they have to be. These blocks are offered to the user at a seperate place from the game board and they can be chosen and placed in a random order as long as they are in the places where they have to be in the end. User is also able to, and usually should, rotate these blocks in order to find the solution for that level. The game consists of different levels which increase in difficulty after each one is completed. There are 3 difficulty levels in the game: Easy, Medium, Hard. Each of these difficulty levels have 3 levels inside meaning, our game consists of 9 levels in total.

1.2. Design Goals

We are required to decompose our system into pieces but before doing that, it is an important aspect for our project to briefly give information about our design goals. These design goals are mostly inherited from the non-functional requirements that we shared in the analysis report that we submitted before.

1.3. End User Criteria:

1.3.1. Simulation of real world

We are implementing a game. Therefore, our main goal is providing entertainment to the user. As one might guess, if we are to offer entertainment, our players should not have difficulties while grasping the basics of game the nor while playing it. With this idea, we have provided user friendly interfaces menus on which the user is able to easily reach wherever they want and perform the desired operations throughout all of the available interfaces. During the gameplay we have decided that our input device to be the mouse. User will be able to easily click and drag the blocks to the necessary places after choosing the right rotation for the blocks, again by using the mouse. This way, we will obtain simplicity for user's benefit.

1.3.2. Basic and applicable:

As in all of the games out there, a player should know how to play the game first. Therefore, we will provide an instructions manual involving how to rotate the blocks, how to move them, what is necessary for a level to be completed. This manual will also contain useful information about the objects of our game.(the thief, police car blocks, map etc.)

1.4. Maintenance Criteria

1.4.1. Modifiability

In our system it needs to be easy to modify the existing functionalities of the system to let it accommodate new ones. More specifically, having completed the first iteration, to realize the changes brought by the second 5 iteration, we aim to make the code “modifiable”. By mentioning “modifiability”, rather than any future development concerns, we are concerned with an implementation which minimizes changes to the existing functions when a new function/change is to be handled. That’s why we are aiming for “modifiable” (independent) code - in case anything goes wrong and this requires changes modifications to existing functions. This can be thought to be more of a precaution to prevent us from rewriting the existing code again in case we fail to integrate new capabilities.

1.5. Performance Criteria

Response Time : For games in general, it is so important that user’s requests and actions have responses as quickly as possible. Otherwise, users might easily lose interest for the game and get distracted. Therefore, we have planned a system that almost immediately responds to the player and also is offering enthusiastic sounds, visual components for entertainment purposes.

1.6. Trade-offs:

1.6.1. Modifiability vs Response Time

Modifiability requires extra calls in order to preserve an environment such that the project can be changed in the future(by adding new features) without decomposing the structure all over again. However, a quick response time requires low call count in order to

instantly respond to user input. Therefore, we will focus on a balance between these two features to both have the user satisfied and make our job easier in the future developments to our game.

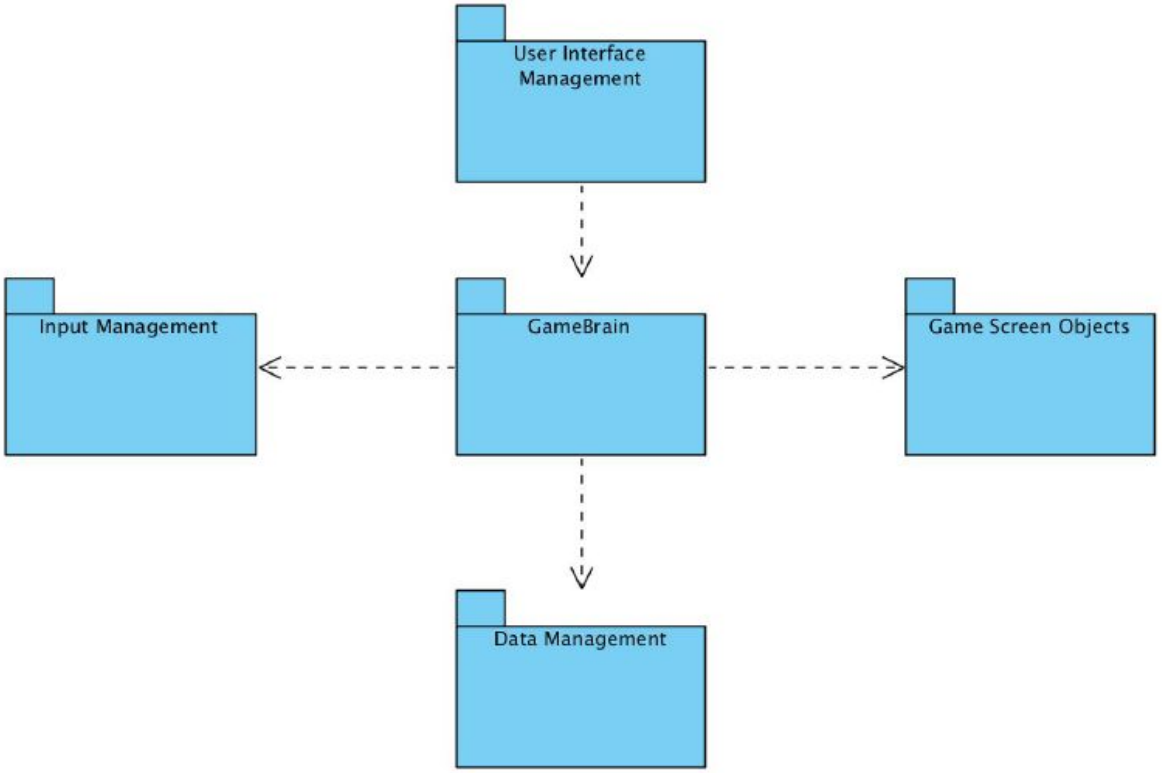
1.6.2. Functionality vs Learning Process (Ease of Use)

For games in general, it is an important feature that the user is able to easily grasp the basics of a game and how to play it. However, it is also expected by most of the customers that games have plenty of features and functionalities in order to “have fun” while playing the game. To conclude, even though these two topics are in conflict with each other, we are trying to offer a game environment that balances these two necessities of a “good” game.

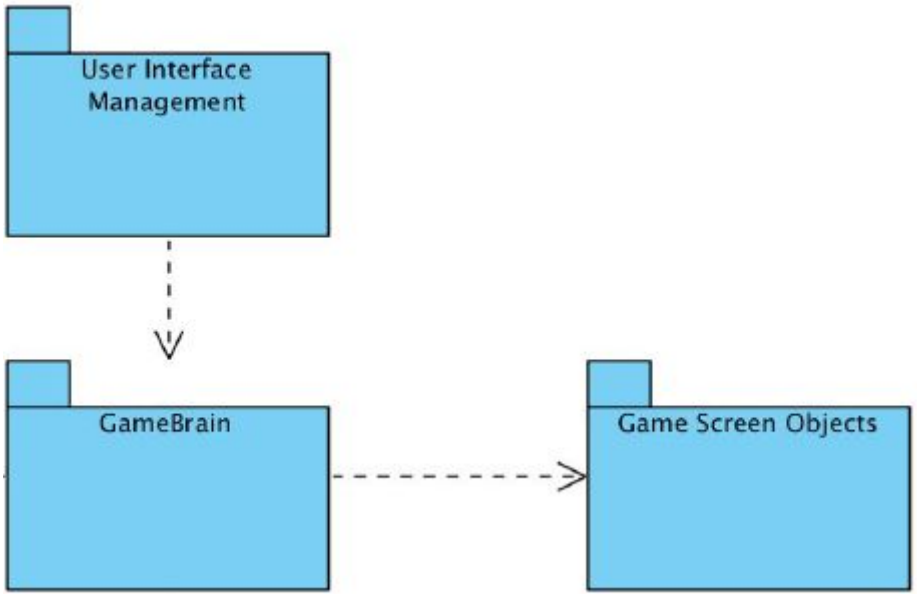
2. High-Level Software Architecture

2.1. Subsystem decomposition

Three-tier architecture is chosen to show structure of the project. Three-tier architecture consists presentation, process logic and data storage. Diagram that shows the project’s three-tier principle is given below as Figure 1. Presentation tier consists user interface and mainmenu, difficulty menu, levelmenu and all of their panels, buttons and components are at this tier. Depending on user’s choices on presentation tier,all signals pass from User Interface Management package to Gamebrain package.



To explain further actions, if the user clicks on the button “Play game” in main menu , program will go to our “DifficultyPanel” to let the user to choose a difficulty. After user’s choice in difficulty menu, levels are seen on the screen and user choice which level he/she want to play.Up to this part the process logic tier is used. Afterwards, with the help of “GameBrain” package and its helpers “Levels”, “Sounds” and “Entities” classes, the game will be set up with its logics. Then, the user will start playing the game. The data storage tier consists of classes that keeps the data needed for objects of the game.These datas are time for point calculation, status of collisions for checking whether game is finished or not, the coordinates of entities such as police, buildings .



2.2. Hardware/software mapping

Software: The game is implemented by using Java packages and libraries. The reason behind the selection of the Java language was its wide usage and compatibility with the all widespread OS variations. As long as Java Runtime Environment is installed, the game can be played on Windows, Mac and Linux platforms.

Hardware: To play the game, in terms of the hardware requirements, the player will need an ordinary computer with a mouse and screen. Furthermore, external audio output device like a speaker or headphones will be needed for the recommended gameplay and hearing the sounds.

2.3. Persistent data management

The game does not involve data storage for the level which user passes .It is because,there is no limitation for selecting levels in order to waste of time in basic levels. In

addition, images of the levels and entities will be stored in a small file that will be in the game's files.

2.4. Access control and security

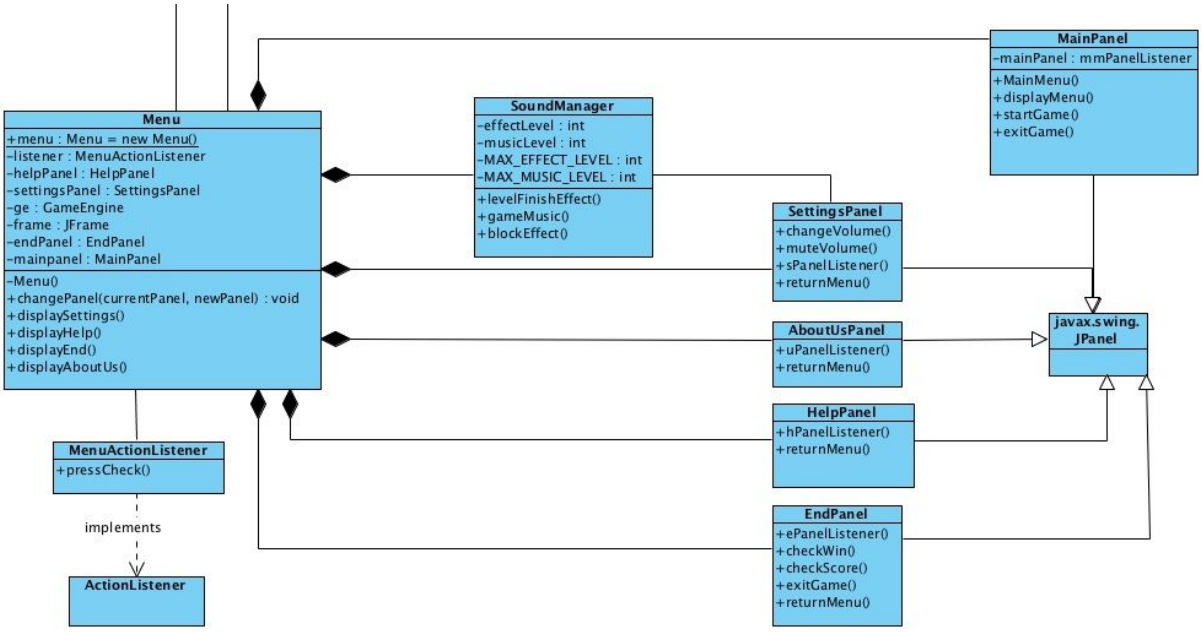
The game does not use any network-related authorization or registration system which will avoid the security issues that may happen because of it. Thats why, no measurement regarding access control and security needs to be taken

2.5. Boundary conditions

The game is to be launched using a ".jar" file. It will not require any prior installation. Also users need the JRE and JVM to execute the game. The game will be easily terminated by the help of "Exit" button on the main menu or X button of the window for the unsafe termination. Unsafe termination will not be affecting any fundamental game structure like source and player will be able to play again the level. Moreover, all menus except main menu have "returntomenu" button. Therefore, user can easily return to menu by clicking that button. When the game finished, user can try again or return to menu. If the game crashes somehow, user can play the same level where he/she trying to finish.

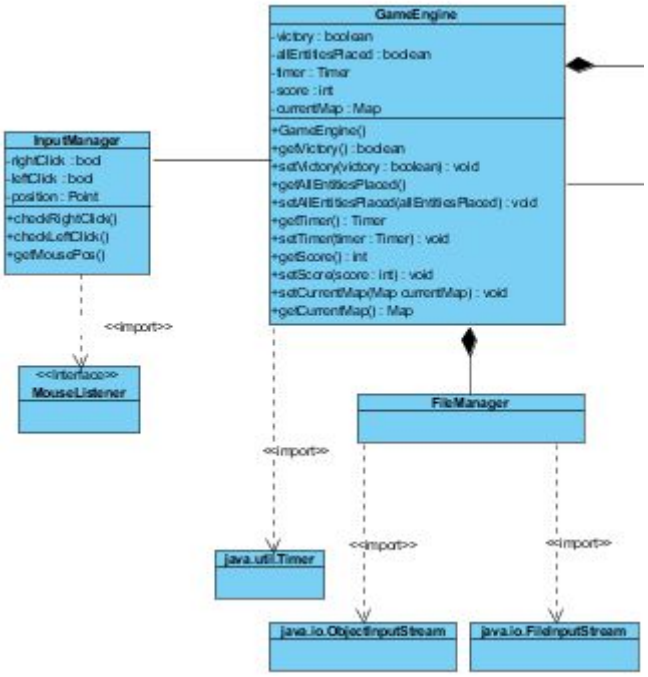
3. Subsystem Services

3.1. User Interface Subsystem



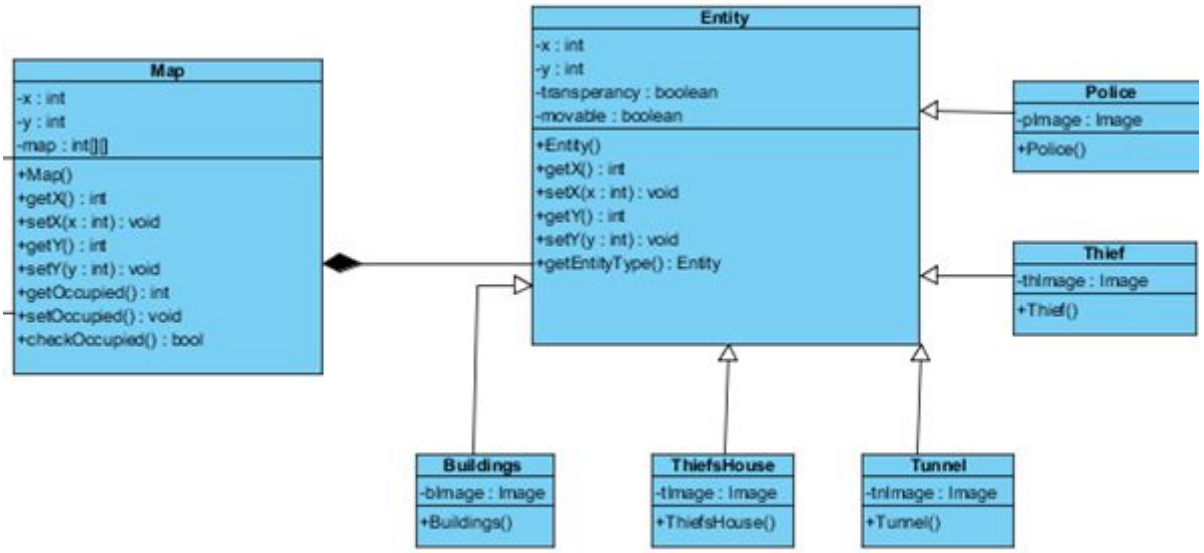
The User Interface (UI) will be in charge of the visual game elements. It will be dealing with the view of the game.This subsystem includes classes that are Menu, MenuActionListener, ActionListener, SoundManager, MainPanel, SettingsPanel, AboutUsPanel, HelpPanel, EndPanel, javax.swing.Jpanel. Menu uses all the panels and SoundManager and check actions of user with the help of MenuActionListenerClass.

3.2. Game Brain Subsystem



Game Brain Subsystem is responsible for creating and updating maps of the game. This subsystem also checks whether game is finished. In addition, all calculations about score of user will be done in Game Brain Subsystem. Furthermore, Game Brain Subsystem uses Filemanager (Data Management) to get levels and InputManager (Input Management) to get mouse movements and clicks.

3.3. Game Screen Objects Subsystem



Game Screen Objects Subsystem decides objects that are shown in the screen while the game starts to running. Map class will draw entities. These entities are Buildings, ThiefsHouse, Tunnel, Thief and Police. This subsystem has also checks whether positions are occupied or not. This check is important when user plays and carry objects because this prevents object conflict.

3.4. Data Management Subsystem

In the project, we are going to use serialization. We will read serialized maps from the hard disk to load the architecture of each level. Instead of initializing whole the map entity array by putting each element on it, we decided to keep these maps as serialized objects in hard disk. For more information, please look at:

https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html

4. Low-level Design

4.1. Object design trade-offs

As our trade-offs were discussed in our “Design Goals” section, we would like to mention solutions that we came up with like dealing with these trade-offs in “low-level” terms.

4.1.1. Response Time vs Modifiability

We are implementing a game.Therefore, response time is one of the most important concerns with our project because we need to respond as quickly as possible to maintain a “fun” environment and prevent possible causes that might lead to loss of interest of the users. One of the ways to do so is, in our object design, providing direct references to objects, especially those who need a faster access time. By this way, we tried to minimize the number of unnecessary calls. These unnecessary calls would cause a slower overall response time especially for the core classes of our game. To also handle with modifiability, we have tried to make our functions as independent as possible from each other. There are of course functions that depend on each other but we have tried to minimize the amount of these functions to improve modifiability for possible changes, adjustments in the future.

4.1.2. Memory vs Response Time

We are aware that memory was not specified in “Design Goals” part of our report. However, consuming more memory means a slowdown for the response time since it conflicts with speed. Since one of this course’s main goals is to learn about and practice with the interactions between the classes, we have not really cared about the memory

consumption. Instead of prioritizing memory, we focused on creating as many classes as possible to practice with the interactions in between them. It is in our attention that our project is relatively a small one so our system will not consume that much of a space in a memory drive. However, we still felt the need to share about the memory planning that we had. In our project, in order to obtain required objects we possess a variety of children classes that can be refactored out to a larger, enumerated type-using, parent class(single).

4.2. Final object design

Game Engine

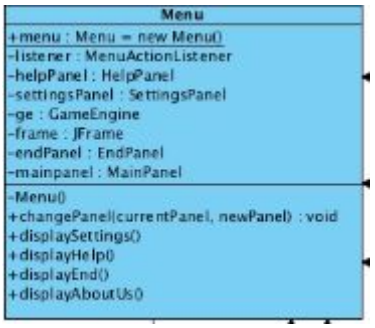
Attributes

- **boolean victory :** This attribute keeps the state of Victory, if all blocks are placed, then the system checks this to determine whether the user wins or loses the game. If true, user wins; if false, user loses.
- **boolean isAllEntitiesPlaced :** The system determines the game over state through this variable. If all entities are successfully placed, this variable becomes true.

- **Timer timer :** This is a single timer, a chronometer. Its considered while the score of the user is calculated at the victory state.
- **int score :** This is the score that is calculated according to the movement number and the time.
- **Map CurrentMap :** This holds the current Map object for the current level.

Functions

- **GameEngine() :** default constructor.
- **getVictory() : boolean** returns victory.
- **setVictory(victory : boolean) :** sets the victory to the parameter.
- **getAllEntitiesPlaced() : boolean** returns the game over state(If all entities placed, then game over).
- **setAllEntitiesPlaced(isAllEntitiesplaced : boolean) :** sets the game over state to the parameter.
- **getTimer() : Timer :**returns the Timer.
- **setTimer(Timer timer) :** sets the Timer to a new Timer instance.
- **getScore() : int** returns the score value.
- **setScore(score : int) :** sets the score value to a new int value
- **getCurentMap() : Map :** returns the current map object.
- **setCurrentMap(Map map) :** sets the current map to a new map.



Menu

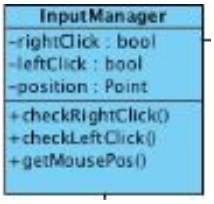
This is the main User Interface.

Attributes

- **+menu : Menu :** This is a *static and public* object that can be reached from other classes. This will be constructed when the program initializes. The reason for this is giving access to Game Brain and Main Frame from each object. For more detailed information, please look at the Java Paterns -> SingleTonPattern.
- **listener : MenuActionListener :** This is a listener that used by Menu object to determine the user action for selecting options.
- **helpPanel : HelpPanel :** This object keeps the help panel.
- **settingPanel : SettingsPanel** This object keeps the settings panel.
- **ge : GameEngine :** This object keeps the game brain.
- **frame : JFrame :** This object keeps the object of main Jframe of the program that is inherited from javax.swing.JFrame class.
- **endPanel : EndPanel :** This object keeps the ending/closing panel
- **mainPanel : MainPanel :** This object keeps the initial panel.

Functions

- **-Menu() :** This is a private constructor that can not be reached from other classes. This constructor initializes the static “menu” object. For more detailed information, please look at the Java paterns -> SingleTonPattern.
- **changePanel(currentPanel, newPanel) : void :** this method changes the on-screen panel. We keep the current Panel for later improvements (Like keeping panels on the stack).
- **displaySetting() :** This function brings the settings panel to the front.
- **displayHelp() :** This function brings the help panel to the front.
- **displayEnd() :** This function brings the ending panel to the front.
- **displayAboutUs() :** This function brings the about us panel to the front.



Input Manager

Attributes

- **rightClick: bool :** Right click of the mouse
- **leftClick: bool :** Left click of the mouse
- **position: Point :** Position of the mouse.

Functions

- **checkRightClick() :** Checks whether right click is clicked or not.
- **checkLeftClick() :** Checks whether left click is clicked or not.
- **getMousePos() :** This function gets the current position of mouse.

Sound Manager

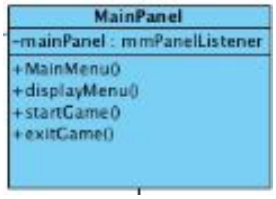
Attributes

- **effectLevel: int :** Level of effects in the game
- **musicLevel: int :** Level of the music in the game
- **MAX_EFFECT_LEVEL: int :** Maximum level of effects in the game
- **MAX_MUSIC_LEVEL: int :** Maximum level of the music in the game

Functions

- **levelFinishMusic() :** This function sets the music when the level is finished either won or lose.
- **gameMusic() :** This function sets the in game music.

- **blockEffects() :** This function sets the effect of the blocks when they are placed.



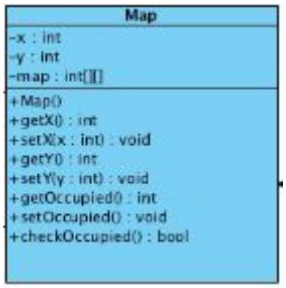
Main Panel

Attributes

- **mainPanel: mmPanelListener :** To listen related user action on main panel.

Functions

- **MainMenu() :** This function creates the main menu.
- **displayMenu() :** This function displays the menu.
- **startGame() :** This function starts the game.
- **exitGame() :** This function provides to exit the game.



Map

Attributes

x: int : x is the vertical coordinate of the map

y: int : y is the horizontal coordinate of the map

map: int[][] : map is the two dimensional array which is vertical and horizontal lengths.

Functions

Map() : Map is the constructor of the Map class.

getX(): int : This function gets the vertical coordinate of the map.

setX(x: int) : void This function sets the vertical coordinate of the map.

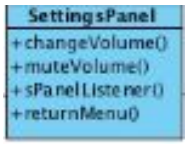
getY() : int : This function gets the horizontal coordinate of the map.

setY(y: int) : void This function sets the horizontal coordinate of the map.

getOccupied() : int This function gets the occupied coordinates of the map.

setOccupied() : int This function sets the occupied coordinates of the map.

checkOccupied() : boolean This function checks whether the specific coordinate is occupied or not.



Settings Panel

Functions

- **changeVolume() :**This method changes the volume of the game.
- **muteVolume():** This method turns off the volume of the game.
- **sPanelListener():** To listen related user action on Menu
- **returnMenu():** This method returns user to the menu.

About Us Panel

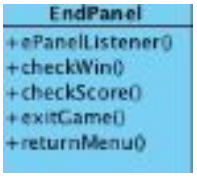
Functions

- **- uPanelListener():** To listen related user action on Menu
- **- returnMenu():** This method returns user to the menu.

Help Panel

Functions

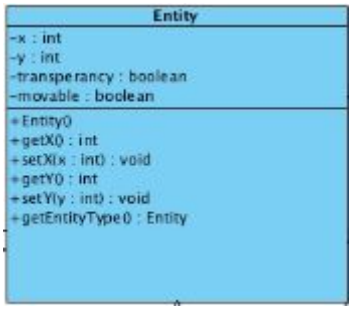
- **hPanelListener():** To listen related user action on menu.
- **returnMenu():** This method returns user to the menu.



End Panel

Functions

- **PanelListener():** To listen related user action on Menu
- **checkWin():** This method checks the whether user wins the game or not
- **checkScore():** This method checks the total score of the game
- **exitGame():** This method exits the game
- **returnMenu():** This method returns user to the



Entity

Attributes

- **x:int and y:int** These coordinates represent the location wherever Entity objects exists.
- **transparency: boolean :** This attribute indicates that whether the objects capable of overcome the obstacles in the game.
- **moveable: boolean: T**his attribute determines the status of the objects whether they are moveable or not

Functions

- **Entity() :** The constructor that will used generating objects
- **getX():int :**This method takes the horizontal coordinate of Map
- **setX(x:int) void :**This method adjusts the horizontal coordinates coordinate of Map
- **getY():int :** This method takes the vertical coordinate of Map
- **setY(y:int)void :**This method adjusts the vertical coordinates coordinate of Map
- **getEntityType():Entity :** This method takes the type of the Entity object type

Buildings

Functions

- **bImage:**Image of the building
- **Buildings():** The constructor that will used generating objects

Thiefs house

Functions

- **tImage:** Image of the thiefs house
- **ThiefsHouse():** The constructor that will used generating objects

Tunnel

Functions

- **tnImage:** Image of the tunnel
- **Tunnel():** The constructor that will used generating objects

Thief

Functions

- **thImage:** Image of the thief
- **Thief():** The constructor that will used generating objects

Police

Functions

- **pImage**:Image of the Police
- **Police()**:**:** The constructor that will used generating objects

4.3. Packages

4.3.1. Java Packages

4.3.1.1. javax.swing

Swing provides the look and feel of modern Java GUI. It is used to create graphical user interface with Java.

4.3.1.2. java.awt

AWT contains large number of classes and methods that allows you to create and manage graphical user interface (GUI) applications, such as buttons, windows, mouse listeners, scroll bars, etc. It is also used to change the color and font of the GUI.

4.3.1.3. java.io

Java IO is an API that comes with Java which is targeted at reading and writing data (input and output). For instance, read data from a file or over network, and write to a file or write a response back over the network.

4.3.1.4. java.imageio

ImageIO is a utility class which provides lots of utility method related to images processing in Java. Most common usage is reading from image file and writing images to file in Java.

4.3.2. Directories

Dividing the system into other subsystem provides a better and uncomplicated work, so MVC Directory Structure is used when our program was developed.

4.3.2.1. Model Directory

This directory contains all related classes for Entity class, and its subclasses.

4.3.2.2. View(GUI) Directory

This directory contains all related classes for the GUI of the program.

4.3.2.3. Image Directory

This directory contains all images used in the GUI such as background, shapes, etc. and the textures of the GUI.

4.3.2.4. Data Directory

This directory contains text files for different prepared levels in the game.

4.3.2.5. Controller Directory

This directory contains management classes for the game such as Game Engine, Sound Manager and Menu.