Bilkent University

CS 319

Object Oriented Software Engineering

RUSH HOUR CC

Design Report

Naisila Puka, Fatbardh Feta, Masna Ahmed, Kunduz Efronova, Talha Zeeshan

Supervisor: Eray Tüzün

November 8, 2018

Table of Contents

1 Introduction

RUSH HOUR CC (Rush Hour by Chain Coders) is a software implementation of the famous sliding block puzzle game Rush Hour, by the group of students called "Chain Coders". We chose this famous board game among all the others because we believe Rush Hour is a good example to start with the basics of object-oriented software engineering.

In this classic remake of Rush Hour, player still has to get one main car out by maneuvering all the other cars out of its way, but we also added some additional features to make it more interesting. Player will have to think hard and plan! The score for each level and challenge will be determined by the number of moves and the time player takes to solve the puzzle and free the car. Game experience will be pleasing and the user will be motivated with additional bonuses like stars, crowns and coins.

This report contains general outline of our project including overview, functional and non-functional requirements, use case, sequence, class, activity and state diagrams. Following that there are mock-ups of user interface giving the taste of a game.

1.1 Purpose of the System

RushHour allows players to solve intriguing puzzles, all within the background of a traffic jam. The game allows players to solve a variety of puzzles of varying difficulty from beginner to advance. The purpose of the game is to allow players to sharpen their puzzle solving ability, by allowing them to solve their way from easy to difficult puzzles. The game also provides the user with vivid graphics and a points-system based on 'stars' and 'coins', thus encouraging them to solve any given puzzle in a minimum number of given moves and time. RushHour is a 2-D game which aims to improve the player's creativity, quick-thinking, and problem solving.

1.2 Design Goals

Before developing the system design and object design, a number of design goals were singled out which would serve as bases of our design plan. Our goal was to design the game in a manner that any user would the game easy to use, easy to run and lacking any sort of inconsistency. Then following requirements were singled out:

1.2.1 Usability Criteria:

1) *Uncomplicated User-Interface*: We aim to present the user, an easy-to-use and aesthetically pleasing user-interface that shall be developed on the basis of lowering the game's complexity. The interface will be simple enough for the user to easily navigate without requiring additional documentation or training.

Furthermore, information regarding the game to be displayed to the user will be kept to a bare minimum allowing the user to play the game without having to worry about trivialities. The user will however will be allowed to view said information separately within the game.

2) *Easy-to-learn*: Gameplay will be developed in a self-explanatory manner. Although instructions will be provided, features of the game will be simple enough for any user playing the game for the first time to understand. Our aim is make the game easily understandable enough that 99% of first-time users will be able to learn the game after solving their first puzzle. The user will, however, be able to view the game's instruction's incase gameplay assistance is needed.

1.2.2 Reliability Criteria:

1) Incorrect Input Handling: Our aim is to develop a game that is free of bugs and glitches. All forms of incorrect input will be handled accordingly to make sure that the user does not have to experience any sort of inconsistent behavior. Events such as unlocking a car that the user cannot afford with the current number of coins he/she possesses, moving a car outside the game grid or in occupied squares, trying to move a vertically oriented car horizontally or a horizontally oriented car vertically, or even trying to access puzzles that are yet to be unlocked among certain incorrect user inputs will be handled by the game to allow a consistent gameplay.

2) *Crash Handling*: Maximum effort will be put into making sure that the game does not crash unexpectedly and in the unlikely event that it does, that the user's data and progress be reloaded once the game restarts. To prevent loss of data, the game will have an auto save feature which will automatically save data after a certain interval if the user is in game solving a particular puzzle. This will allow the user to continue with his progress, after restarting the game post-crash.

1.2.3 Performance Criteria:

1) *Refresh Rate:* We aim to provide the user with the smoothest experience possible at the level of java GUI. A smooth experience requires a refresh rate of at least 60 (refreshes per second) and at 60 frames from second, therefore, we will make substantial effort to make sure that the game refreshes at least 60 times per second to make sure that that user generated movements within the game are smooth and accurate. We will also attempt to make the game as little CPU-intensive as possible to make sure that even the most basic computer system will be able to support a respectable frame rate for the game.

2) *Response Time*: Our goal is to make sure that the user does not feel any disparity between any movements he makes in the game and the response on his screen. By optimizing certain features of the game such as refresh rate, fps (frames per second), and by using lower order algorithms, we will attempt to

make sure that the game's response time does not exceed 2 seconds at the very worst.

3) *Storage*: We aim to use minimal storage for the game. The game will use at most 5 basic text files, each not exceeding a size of 1000 kilobytes. The files will contain basic data regarding the user's personal information, the user's progress within the game, temporarily backed up data, and information regarding the arrangements of the various puzzles.

1.2.4 SUPPORTABILITY CRITERIA:

1) *Setup*: We attempt to make sure that the user does not have to experience lengthy installation procedures to be able to play the game, therefore the game will be available for download as an executable JAR (Java Archive) file since we will develop the game in the Java Programming language.

2) *Support for older Operating Systems*: We aim to use Java SE 8 for implementing the project. Java SE 8 is currently the recommended and supported version of the deployment stack [reference] therefore the game should be able to run on relatively older operating systems.

1.2.5 REQUIREMENT PRIORITIES AND THEIR EFFECT ON FUNCTIONALITY:

The usability criteria of the game will be given the highest priority during development since we aim to make sure that the user does not find any sort of difficulty while learning the game. This means that if a certain additional feature of the game does not meet our usability criteria it not be included as part of the final game. We aim to make the game as easily learnable and playable as possible which means that complex features for the game are not viable.

1.2.6 PERFORMANCE OPTIMIZATION:

Performance is second highest priority for the functioning of the game. While refresh rate, response time and storage optimizations will be essentially implemented, we aim to use further techniques to improve the game's performance, while keeping check on memory limitations. One such technique we aim to use is to keep the game's "Main Menu" interface, running at all times. While this may lead to additional memory usage, it will lead to the user having a better experience while transitioning between the various puzzles when interaction with the main menu becomes necessary. We may use the Java JPanel object's visibility attribute to attempt this if suitable.

2 System Architecture

2.1 SUBSYSTEM DECOMPOSITION

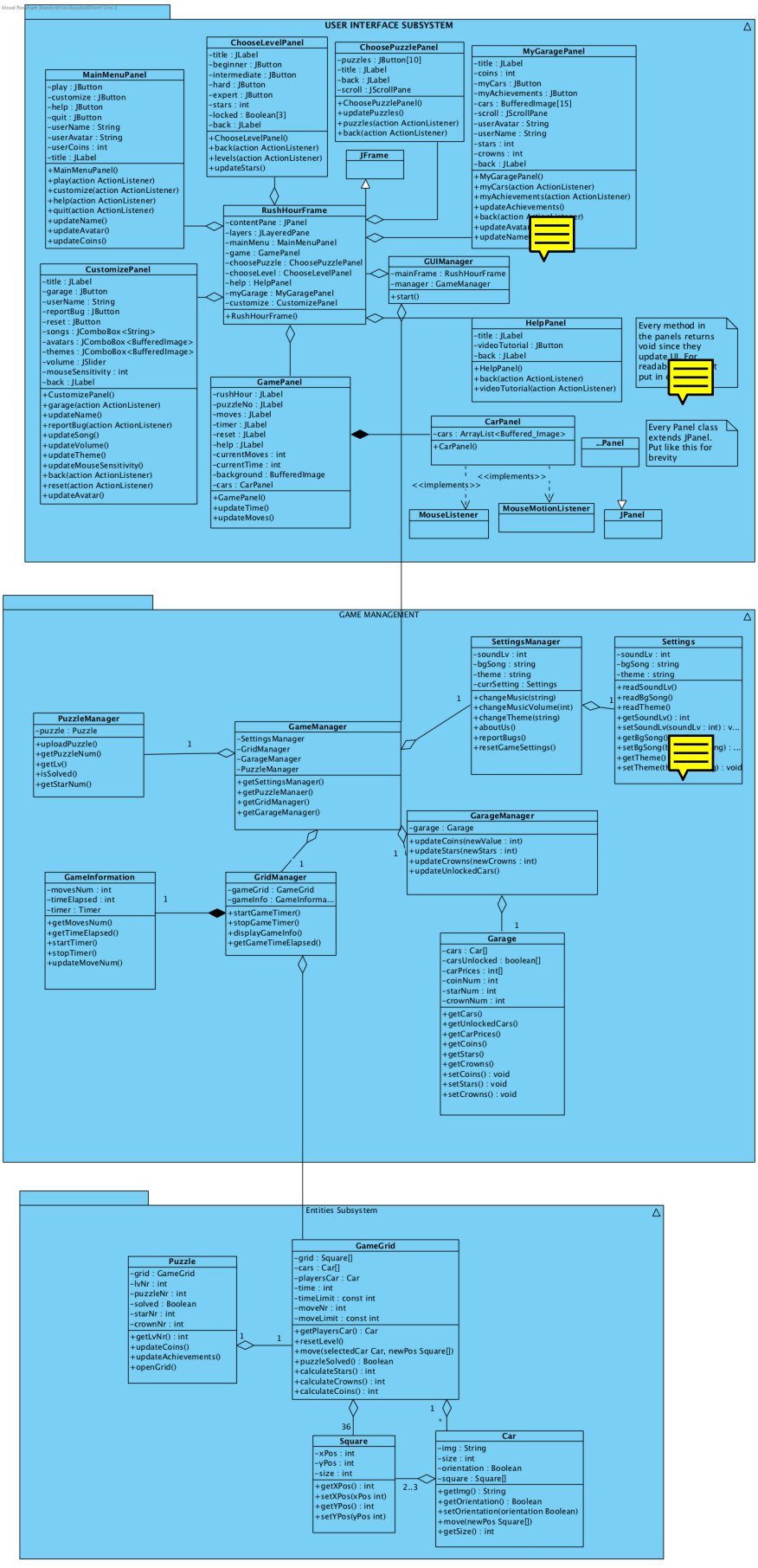

FIGURE 1 SYSTEM DESIGN DIAGRAM

2.2 ARCHITECTURAL STYLES LAYERS

2.2.1 LAYERS

To better illustrate the systems of Rush Hour, the layered architecture pattern will be used. The benefit of using layered architecture design is that it allows us separate the functionality and concerns of components in the system. Components within a specific layer deal only with logic that pertains to that layer. Each layer of the layered architecture pattern has a specific role and responsibility within the application and forms an abstraction around the work that needs to be done to satisfy a particular request. Changes made in one layer of the architecture will not affect the other layers which means that each layer is independent of the other layers, thereby having little or no knowledge of the inner workings of other layers in the architecture.

Using layered architecture design, the game "Rush Hour" has been divided into three layers to better show the entire system structure of the game. The first layer handles all user interface related logic. This layer will have the highest hierarchy in our design. The second layer handles all logic related to the running and management of the game. The third and last layer handles logic of all objects and entities that are part of the game.

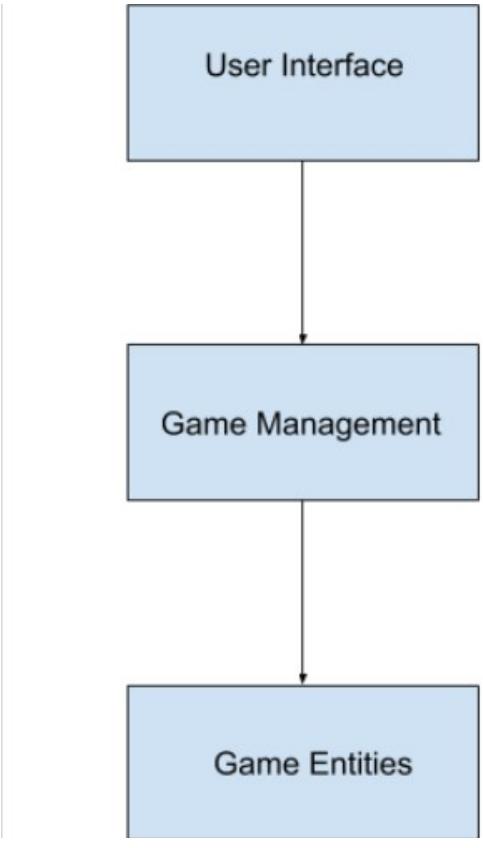

FIGURE 2 ARCHITECTURAL LAYERS

2.2.2 Model View Controller (MVC)

MVC is an architectural pattern that divides an application into three main logical components; The Model, the view and the controller. Each component handles specific development and functional aspects of an application.

- The Model

The model component will correspond to all game logic that the user will work with. In Rush Hour, the model component will be composed of the Game Entities layer since data will be generated by the user interacting with the objects and entities in that layer (i.e: movement of the car horizontally by x boxes).

- The View

The view component in our application will consist of the user interaction components. This will correspond to interaction between the user and the system via the use of menus and buttons. The controller component interacts with the view component to render the final output after user interaction (i.e: showing updated location of car after it has been moved).

- The Controller

The controller component acts as an interface between the view and the model components. In this application, the controller component will be composed of the Game Management layer. This component will process all logic coming from the model component and interacts with the view component to display its final output. Hence, in this application, the game management layer (the controller) will process logic generated by the moves the user makes with the objects available to them (the model) and display the final output that relate to the logic generated (the view).

2.3 Hardware/Software Mapping

We are going to use Java to implement our project while for the GUI components of the game we will be using the JavaFX framework. Since Java is a programming language that provides cross-platform portability it will work on all platforms that have a JVM installed, making our program suitable for all major operating systems (Windows, MAC OS, Linux).
The hardware components that the system will interact with are the mouse, keyboard and speaker. The keyboard will be used to input users name while the mouse will be used to drag the cars during gameplay as well as interaction with the system. The system will also use the speakers to play the sound effects and

the music of the game. To be able to use those hardware the user is supposed to have installed all the appropriate drivers and also use a compatible version of Java(recommended JDK1.7).
Our game will be an offline game as it will not require internet connection. All the data that our game will save are stored directly in the machine hard disk.

2.4 Persistent Data Management

Rush Hour will use .txt files to read and save data into the machine. Puzzle objects will be written during game development and stored in JSon format while they will be read by the game every time they are needed during gameplay. Other data like Game Settings, achievement's , coin number, avatar properties and more will be saved everytime the user changes or updates them. The system reads those files anytime and updates the properties of the game according to the data found in the .txt files.

2.5 Access and Security System

Rush Hour is implemented in the way that it will not require any kind of network connection. All necessary information for gameplay will be held in .txt files instead of database, and, therefore, there will not be any restrictions or control for access.User profiles will contain only name and avatar, no password will be included, hence there will be none of security issues in Rush Hour.

2.6 Boundary Conditions

2.6.1 Execution

Rush Hour does not require any software other than Java Runtime Environment installed on the computer.

2.6.2 Initialization and Termination

Rush Hour will come with an executable .jar file. The user can initialize the program by clicking the executable file. The program will be terminated when player quits the game by clicking "QUIT" button. If player wants to quit during game play, we provided "BACK" button which will throw user to main menu where player can end game using "QUIT" button. Rush Hour will be full screen game, therefore, there will not be "x" button.

2.6.3 Exception Handling

The game highly like cannot result in exceptions except I/O hardware since it does not have any database or network connection that. However, if there is an exception in the initializing of the persistent objects or the initialization of the

game or the connection exception due to I/O hardware, the system will display an error message that shows the exception and all exceptions will be accordingly handled.

3 Subsystem Services

3.1 User Interface Subsystem

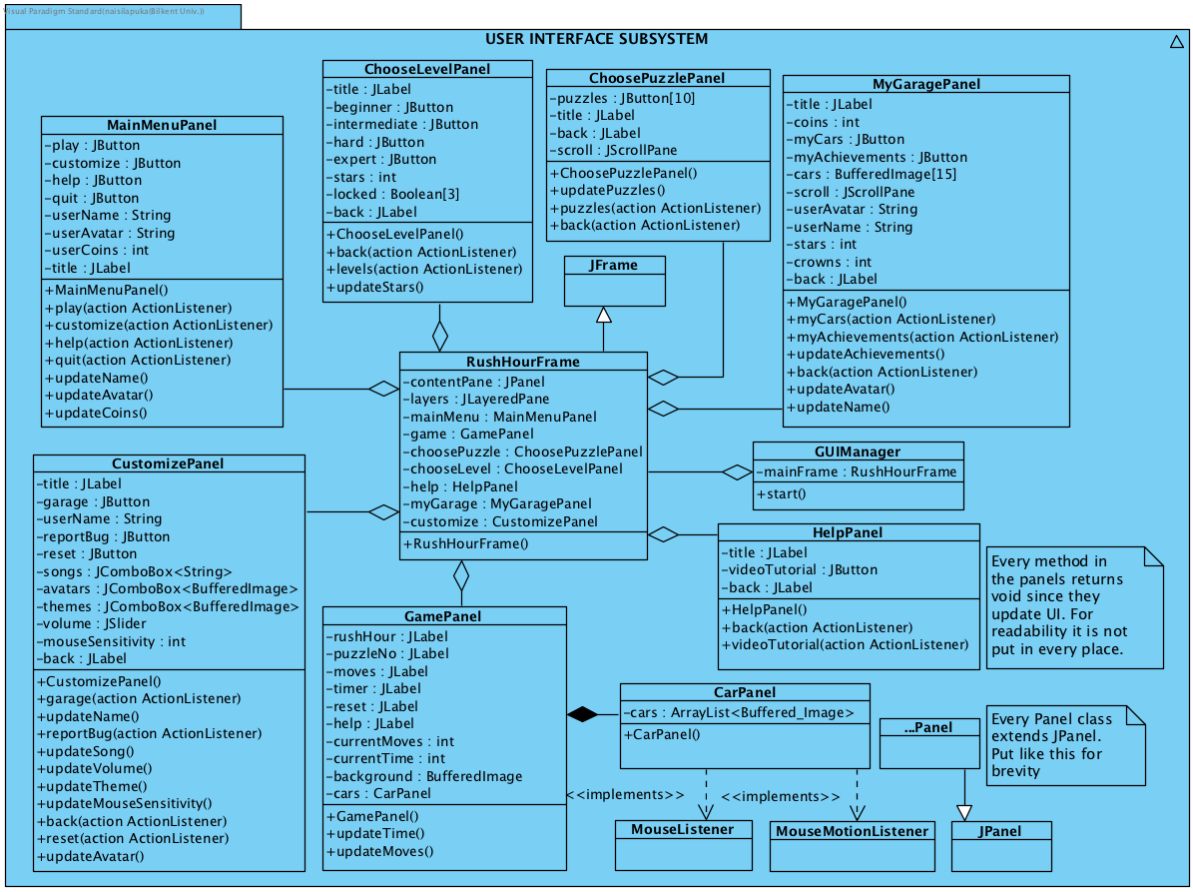

FIGURE 3 USER INTERFACE SYBSYSTEM DECOMPOSITION

3.1.1 RushHourFrame

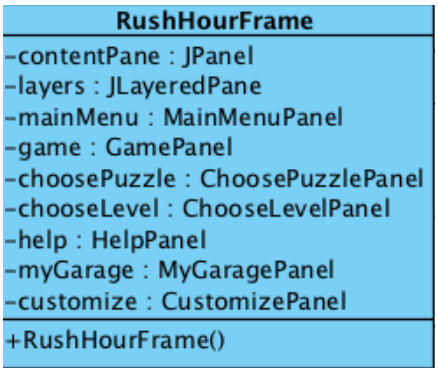

FIGURE 4 RUSH HOUR FRAME

Attributes:

private JPanel contentPane: This will represent the current panel displayed in the frame

private JLayeredPane layers: through this attribute, all game panels will be layered, and the one on top will be displayed

private MainMenuPanel: the frame of the application will have this attribute in order to display the menu panel and update it accordingly by taking information from the other panels as well.

private GamePanel game: this panel will display the current game being played and will be updated accordingly in the main frame

private ChoosePuzzlePanel choosePuzzle: this panel will display the choice of the 4 levels, and based on the number of stars, some of them will be locked

private ChooseLevelPanel chooseLevel: similar to choosePuzzle

private HelpPanel help, private CustomizePanel customize, private MyGaragePanel myGarage: through these attributes the user will be able to switch between layers of the game.

Constructor:

RushHourFrame: In the constructor all game panel navigation functionalities are given to different buttons in the panels in the JLayered Pane

3.1.2 MAIN MENU PANEL

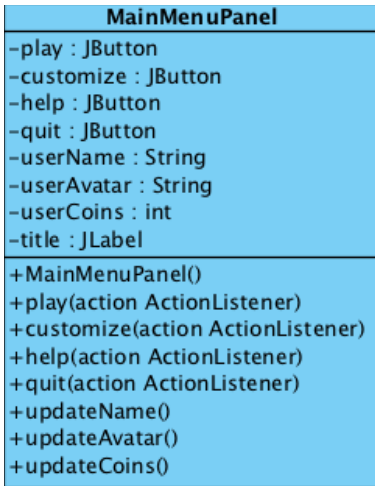

FIGURE 5 MAIN MENU PANEL

Attributes

private JButton play, customize, quit, help: these buttons serve for depicting the navigation to other game panels

private JLabel title: this is the formatted title of the panel

private String username, userAvatar : these strings will contain the String path to the specified image and the user name

private int userCoins: this integer will hold the current numbe of coins of the player so that they are always showed

Constructor:

MainMenuPanel() This sets up the main menu panel with all its attributes in the specified place

Methods:

public void play/customize/help/quit(action ActionListener) these methods add the action that the buttons listen to when they are pressed. The method will be called from RushHourFrame constructor.

public void updateName/Avatar/Coins() these methods will update the coins and the user information that appears on game launching, based on the user's progress and his preferences on his profile set at Customize option.

3.1.3 CUSTOMIZEPANEL

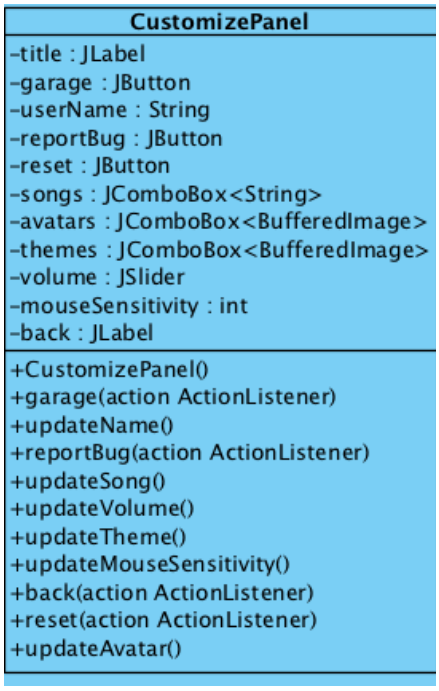

FIGURE 6 CUSTOMIZE PANEL

Attributes:

private JLabel title:

private JButton garage: Button that calls the garage panel and shows it. Calls garage() method.

private String userName: Holds the name of user.

private JButton reportBug: Calls the reportBug() method.

private JButton reset: Button that calls the reset method.

private JComboBox<String> songs: Box with all the songs that the system can play.

private JComboBox<BufferdImages> avatars: Similar to the songs JComboBox contains all the avatar images that the player can choose.

private JComboBox<BufferdImages> themes: Contains all the themes that the player can apply on the game.

private JSlider volume: A JSlider that is used to change the sound volume.

private JLabel back: A JLabel that will have an action listener that calls return() method when clicked.

Constructor:

public CostumizePanel(): It initializes the CostumizePanel.

Methods:

private void garage(ActionListener): Opens MyGaragePanel.

private void updateName(): Updates player name label.

private void reportBug(action ActionListener): opens the report bug panel to provide comunication between the user and the developers.

private void updateSong(): Allows the user to change game song.

private void updateVolume(): Allows the user to change game song volume.

private void updateTheme(): Allows the user to change game theme.

private void updateMouseSensitivity(): Allows the user to change the sensitivity of the mouse.

private void back(action ActionListener): Returns the user to the MainMenu Panel.

private void reset(action ActionListener): Resets the Settings object to the original state.

private void updateAvatar(): Changes the game avatar to the new one selected by the user.

3.1.4 CLASS HELPPANEL

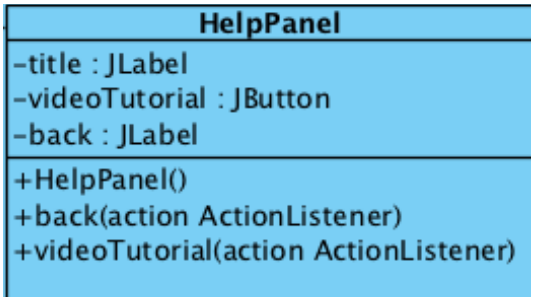

FIGURE 7 HELP PANEL

***Attributes*:**

private JLabel title: this attribute provides a text for the title of the HelpPanel

private JButton videoTutorial: this button will allow functionality for the user to be able to open a panel displaying a video tutorial for the game

private JLabel back: this label will provide functionality for the user to go back to the previous panel

***Constructor*:**

public HelpPanel(): this will initialize the information to be displayed on the help panel

***Methods*:**

public void back(ActionListener listener): this method will be linked to the back label to provide functionality to return to the previously displayed panel

public void videoTutorial(ActionListener listener): this method will provide functionality to the videoTutorial Jbutton and display a video containing instructions regarding the game

3.1.5 GAMEPANEL

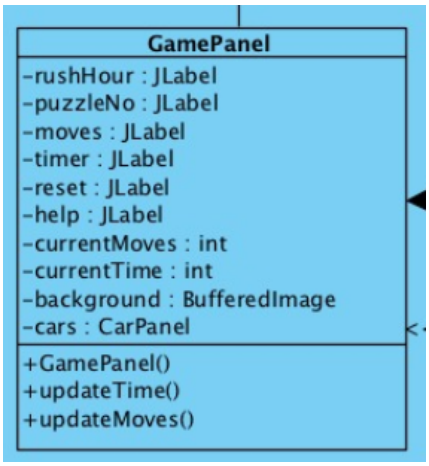

FIGURE 8 GAME PANEL

Attributes:

private JLabel rushHour: The title of the game is shown during gameplay.

private JLabel puzzleNo: The number of the puzzle is shown in this label.

private JLabel moves: Shows the number of moves the player can make to get three stars for that level.

private JLabel timer: Shows the timer the user should take to get stars.

private JLabel reset: Resets the puzzle, time and count number.

private JLabel help: Shows instructions of the game.

private int currentMoves: Counts the movements the user has made trying to solve the puzzle.

private int currentTime: Shows the time the user is taking to solve the puzzle.

private BufferedImage background: Has the background of the game.

private CarPanel cars: Contains the car images of the game.

Constructors:

public GameManager(): it initializes the GameManager object .

Methods:

private void updateTime(): While the user is playing the game this method updates the time he/she takes to solve the puzzle.

private void updateMoves(): After every move the player does this method updates the moveCount. This method counts the moves the user makes until he/she solves the puzzle.

3.1.6 CHOOSE LEVEL PANEL CLASS

FIGURE 9 CHOOSE LEVEL PANEL

Attributes:

private JLabel title, back: a label for the Jbuttons the user will interact with.

private JButton Beginner, intermediate, hard, expert: Jbuttons that the user will interact with when choosing levels.

private int stars: the total number of stars the user currently has.

private boolean[3] locked: a boolean of array of size three that locks/unlocks the intermediate, hard and expert levels after certain conditions are met.

Constructor:

public ChooseLevelPanel( ): A constructor for the choos LevelPanel. Initializes the choose level panel with values read from the saved text files related to the game.

Methods:

public void back(action ActionListener): add an actionListener to the "back" JButton which determines to which panel the UI will go "back" to.

public void levels(action ActionListener): adds an actionListener to all four level buttons when they are not locked. Levels corresponding to the selected buttons are then presented to the user (communications occurs through the RushHourFrame class).

public void updateStars( ): updates the number of stars the user has after they have used them to unlock harder levels.

3.1.7 CHOOSE PUZZEL PANEL CLASS

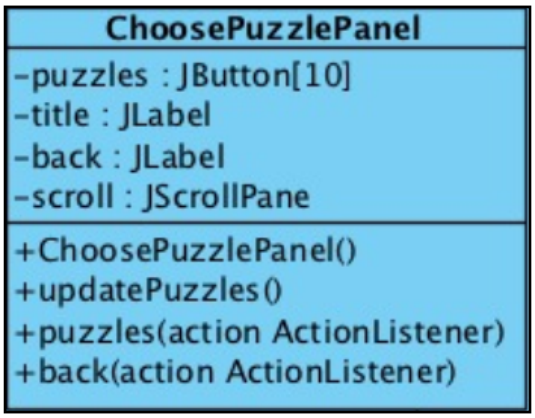

FIGURE 10 CHOOSE PUZZLE PANEL

Attributes:

private JButton[10] puzzles: An array of size 10 of JButtons that correspond to puzzles present in a selected difficulty level. Each difficulty level contains 10 puzzles.

private JLabel title, back: Labels for the puzzle and back buttons present on the screen.

private JScrollPane scroll: scrollPane for scrolling the puzzle selection grid.

Constructor:

public ChoosePuzzlePanel(): Initializes a ChoosePuzzlePanel object with attributes valued assigned from saved text file.

Methods:

public void updatePuzzles(): updates the stars, crowns, coins, moves and time taken properties of the puzzle and saves them. The user will be able to see how they did at their previous attempt at the puzzle and improve the score if they wish. If the new score is better than the previous score, this function will be called to update the score puzzle.

public void puzzles(action ActionListener): adds actionListener to puzzle buttons so that they can be selected by the user.

public void back(action ActionListener): adds actionListener to the back button to be able to go back to the previous panel.

3.1.8 MyGaragePanel

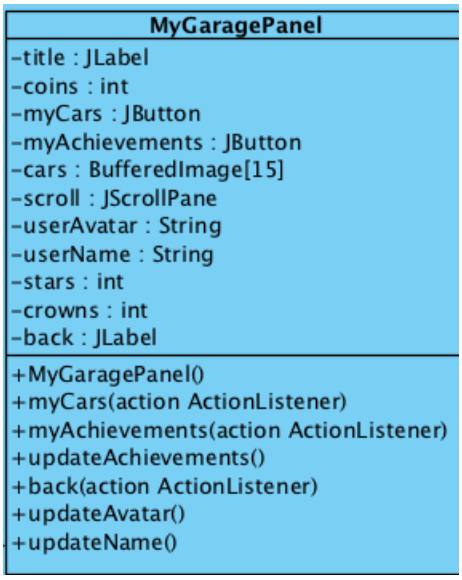

FIGURE 11 MY GARAGE PANEL

"My Garage" is a place where player will be able to view cars and achievements, manage his cars and unlock new cars according to his achievements. MyGaragePanel will nicely display all features of My Garage using GUI.

Attributes:

private Jlabel title: This attribute will be used to display title of panel. Title, obviously, will be "My Garage".

private int coins: Attribute to keep amount of coins user has.

private JButton myCars: Attribute JButton with name "MY CARS" which will have actionlistener and will display cars when pressed.

private JButton myAchievements: Attribute JButton with name "MY ACHIEVEMENTS" which will have actionlistener and will display achievements when pressed.

private BufferedImage[15] cars: This attribute will allow us to view images of all cars, locked and unlocked that the user will be able to access in gameplay.

private JScrollPane scroll: Attribute JScrollPane is used by user to scroll through cars in My Garage.

private String userAvatar: Attribute string to represent image of user.

Private String userName: Attribute string to represent name of user.

private int crowns: Attribute to represent number of crowns user has.

private int stars: Attribute to represent number of stars user has.

private JLabel Back: Attribute JLabel with name "BACK" which will have actionlistener and will redirect user back to Settings panel.

Constructors:

MyGaragePanel(): Constructor to create instance of MyGaragePanel.

Methods:

public void myCars(action ActionListener): Method adds actionlistener to My Cars JButton in order to get notified when user presses My Cars button.

public void myAchievements(action ActionListener): Method adds actionlistener to My Achievements JButton in order to get notified when user presses My Achievements button.

public void updateAchievements(): This method will update the user's achievements after the end of a particular game

public void back(action ActionListener): Method adds actionlistener to BACK JButton in order to get notified when user presses BACK button.

public void updateAvatar(): This method will allow the user to update his/her avatar by interacting with the respective manager classes

public void updateName(): This method will allow the user to change his/her username. The changes will be reflected in the game's user information text file

3.2 GAME MANAGEMENT SUBSYSTEM

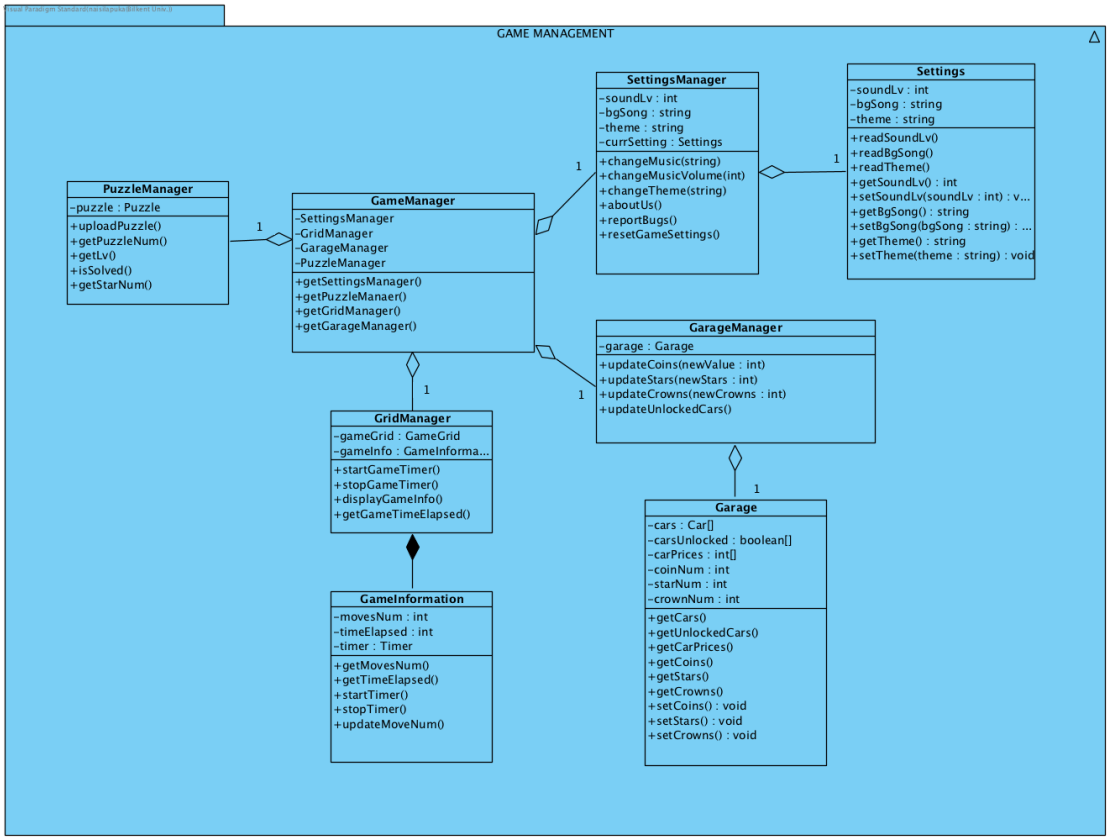

FIGURE 12 GAME MANAGEMENT SUBSYSTEM DIAGRAM

3.2.1 GAMEMANAGER CLASS

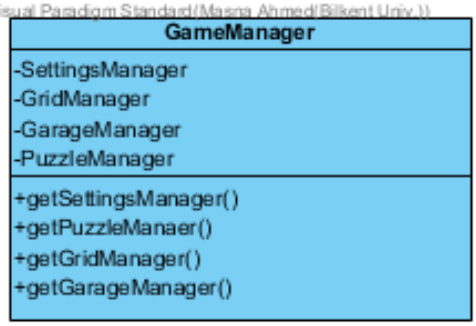

FIGURE 13 GAME MANAGER

***Attributes*:**

private SettingsManager settingsManager: a settings manager object that will be called to initialize and manage the game's settings

private GridManager gridManager: a gridManager object that will be instantiated to develop the game grid and handle the entirety of its workings, including the current game's information

private GarageManager garageManager: an object that will be used to initialize a garage object and handle its inner workings

private PuzzleManager puzzleManager: this instance will manage the type of puzzle that is to be played by the user and procedures such as loading a given puzzle and storing information regarding a solved puzzle

***Methods*:**

public SettingsManager getSettingsManager(): this method returns the class attribute SettingsManager

public GridManager getGridManager():this method returns the class attribute SettingsManager

public GarageManager getGarageManager():this method returns the class attribute GarageManager

public PuzzleManager getPuzzleManager():this method returns the class attribute PuzzleManager

3.2.2 GRIDMANAGER CLASS

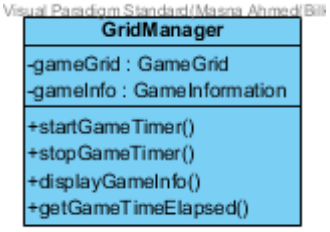

FIGURE 14 GRID MANAGER

***Attributes*:**

private GameGrid gameGrid: an instance of the game entity, GameGrid, which serves as the major controller of the game

private GameInformation gameInfo: an instance of a GameInformation class which acts as the current game's statistics keeper

***Methods*:**

public void startGameTimer(): starts the timer of the gameInfo attribute

public void stopGameTimer(): stops the timer of the gameInfo attribute

public void displayGameInfo(): this method will be used to display all information relevant to a current game to the user

public int getGameTimeElapsed(): this method will be used to get the time in seconds that has been elapsed since the start of the, provided by the gameInfo attribute

3.2.3 GAMEINFORMATION CLASS

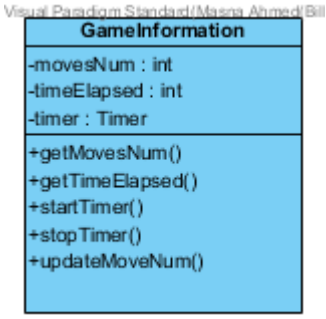

FIGURE 15 GAME INFORMATION

***Attributes*:**

private int movesNum: this attribute will be used to keep track of the number of moves (a single movement of a single car) made by the user since the start of the game

private int timeElapsed: this attribute will be used to keep track of the amount of time in seconds that have elapsed since the start of the game

private Timer timer: a timer object use to handle the time keeping of the game

***Methods*:**

public int getMovesNum(): a getter method for the movesNum attribute

public void updateMovesNum(): this method will be used to increment the number of moves everytime they are made

public void startTimer(): starts the timer object and marks the start of the game

public void stopTimer(): stop the timer object and marks the end of the game

public int getTimeElapsed(): a getter method for the timeElapsed attribute

3.2.4 GARAGE CLASS

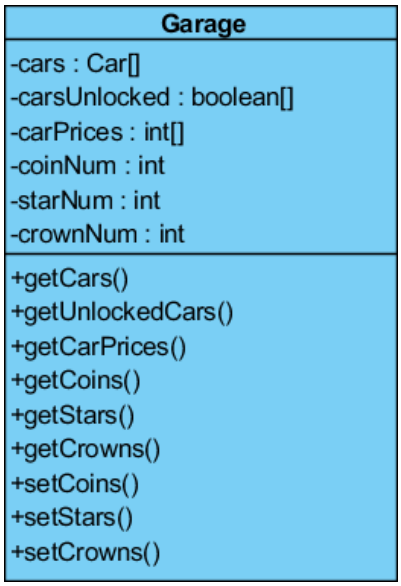

FIGURE 16 GARAGE

The Garage class handles the logic associated with the "My Garage" use case. This class keeps track of the total number of car models available, the total number of car models already owned by the player, the models available for purchase by the user, the cost of buying each model and the stars and crowns the user has.

Attributes:

private Car[ ] cars : An array of car objects that stores all models owned by the user.

private boolean[ ] carsUnlocked: An array of boolean values that works in tandem with the array of cars to determine whether they are available to the user for purchase or not. For example boolean value at index 3 relates to car at index 3 in car objects array.

private int[ ] carPrices : An array of integers that stores the price for unlocking available car models. This array works in tandem with the boolean and car objects array for transaction.

private int coinNum: Number of coins the user has accumulated.

private int starNum: Number of stars the user has accumulated.

private int crownNum: Number of crowns the user has accumulated.

Methods

public car[ ] getCars(): Returns the array of car objects that contains all car models in the game to the game manager. Allows us to update the availability of the car using two methods given below.

public int[ ] getUnlockedCars(): returns array of integers that contains the index of car objects that are unlocked hence available for purchase.

Public int[ ] getCarPrices(): returns an array of integers that contains the prices of each car object in cars array.

public int getCoins(): returns the total number of coins the user has.

public int getStars(): returns the total number of stars the user has.

public int getCrowns(): returns the total number of crowns the user has.

public void setCoins(): changes the number of coins the user has if user uses them for purchase of a new model.

public void setStars(): changes the number of stars the user has if user uses them for purchase of a new model.

public void setCrowns(): changes the number of crowns the user has if user uses them for purchase of a new model.

3.2.5 GARAGE MANAGER CLASS

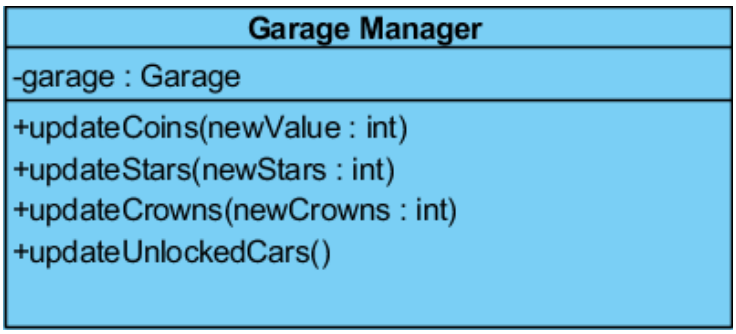

FIGURE 17 GARAGE MANAGER

The Garage Manager class is a facade class that connects the user interface subsystem and the garage class. It handles all logic related to communication between the user and the garage class and updates the attributes of the garage according to the users actions.

Constructor:

public GarageManager(): Initializes a GarageManager object with attribute assigned from saved text file if game has been played before.

Attributes:

garage Garage: An object (instance) of the garage class.

Methods:

public void updateCoins(int newValue): updates the number of coins the user has if user has a bought a new car model. Garage manager uses the set and get methods of Garage to update the value.

public void updateStars(int newValue): updates the number of stars the user has if user has a bought a new car model. Garage manager uses the set and get methods of Garage to update the value.

public void updateCrowns(int newValue): updates the number of crowns the user has if user has a bought a new car model. Garage manager uses the set and get methods of Garage to update the value.

public void updateUnlockedCars( ): updates the number of cars available for the user to buy. Makes locked cars

3.2.6 PUZZLE MANAGER

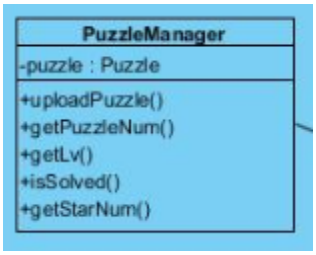

FIGURE 18 PUZZLE MANAGER

Attributes:

private Puzzle puzzle: this attribute holds a Puzzle object with a specific grid configuration.

Constructors:

public puzzleManager(): it initializes the Puzzle Manager object .

Methods:

private void uploadPuzzle(): this method is used to read a Puzzle object from the .txt file and restore it. The Puzzle object that is saved has custom features that were modified and written back into the .txt file.

private void getPuzzleNr(): after the Puzzle object is read from the .txt file it can be asked to return the difficulty number(id) of the puzzle.

private int getStarNr(): return the amount of stars that the user got the last time he/she played that puzzle.

private int getPuzzleLv(): return the lv of the puzzle the user is interacting with.

private boolean isSolved(): this method checks the .txt file if the selectd puzzle has been solved before.

3.2.7 SETTINGS MANAGER:

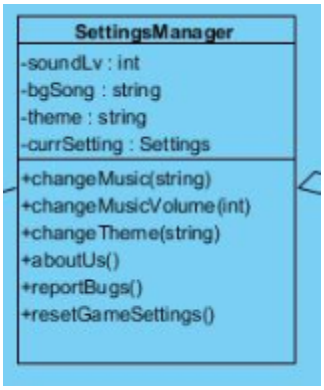

FIGURE 19 SETTINGS MANAGER

Attributes:

private int soundLv : saves the sound lv of the system.

private String bgSong: has the URL addres of the mp3 file.

private String theme: is the id of the theme of the game.

private Settingz currSettings: contains the current Setting object that specifyes the running settings of the game.

Constructors:

public settingsManager(): builds the settings object of the game according to the previously saved settings object in the .txt file.

Methods:

private int changeMusicVolume(int volume): changes the volume of the music in the settings object.

private void changeMusic(String newMusic): sets the music tune of the settings object to the audio file shown in the newMusic URI.

private void changeTheme(String themeName): changes the theme of the game according to the themeName selected by the user.

private void aboutUs(): shows info about the developers.

private void reportBugs(): shows a window that allows the user to contact directly with the developers.

private void resetGameSettings(): resets all the properties of the game Settings object to their initial state.

3.2.8 SETTINGS :

FIGURE 20 SETTINGS

Attributes:

private int soundLv: contains the current sound lv of the game.

private String bgSong: contains the URL address of the song as a String .

private String theme: contains the name of the current theme.

Constructors:

public settings(): constructs the default settings object of the game .

Methods:

public int readSoundLv() : read the sound lv of the settings object saved in the .txt file.

public readBgSong(): read the background song of the settings object saved in the .txt file.

public readTheme(): read the theme name of the settings object saved in the.txt file.

public int getSoundLv(): returns the sound lv of the game .

public void setSoundLv(int soundLv): sets the sound lv equal to soundLv .

public string getBgSong(): returns the song that is playing during the game .

public void setBgSong(String bgString): changes the background song to the song addressed throught the URI of bgString.

public String getTheme(): returns the name of the theme that is being used.

public void setTheme(String theme): changes the theme in the settings object .

3.3 ENTITIES SUBSYSTEM

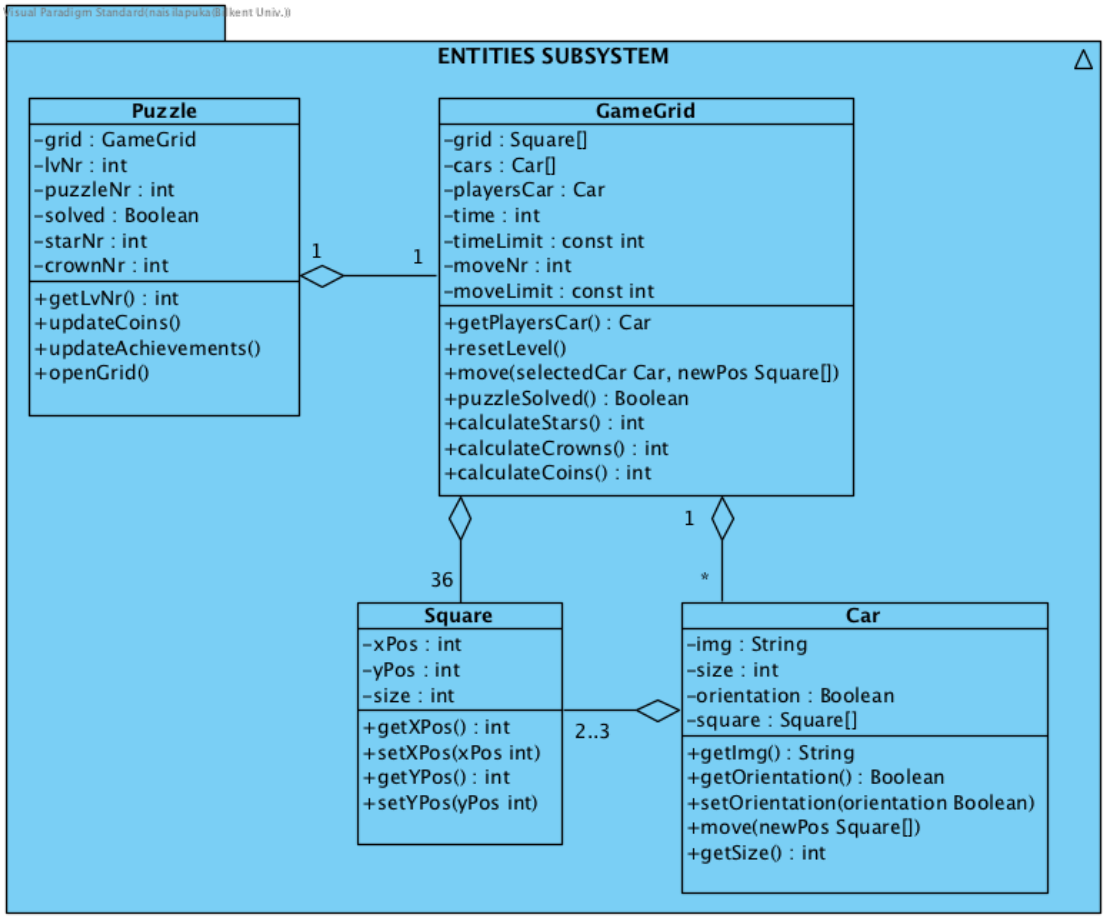

FIGURE 21 ENTITIES SUBSYSTEM

3.3.1 GAMEGRID

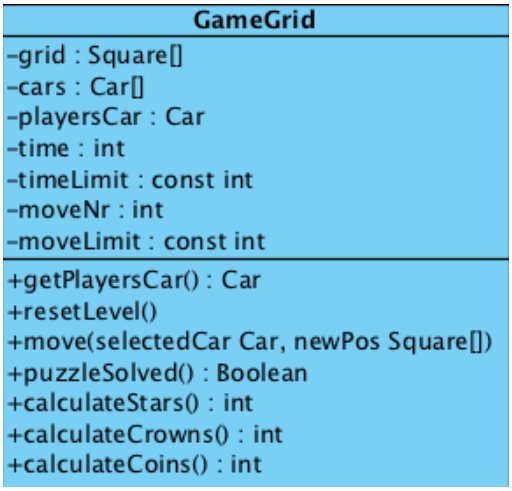

FIGURE 22 GAME GRID

The class which represents the game grid consisting of array of squares. Game grid will have particular number of cars for each puzzle and an exit. Game Grid will be responsible for movements of car, number of movements, time spent to solve a problem and achievements part.

Attributes:

Private Square[ ] grid: Attribute that represents grid of game consisting of array of Square objects.

Private Car [ ] car: Attribute to keep cars that need to be moved in order to get main car out. Represented by array of Car objects.

Private Car playersCar: Attribute to represent main car (car of player) which needed to get out in order to win game.

Private int time: Attribute to keep time spent by player for solving particular puzzle.

Private const int timeLimit: Constant attribute to represent time allowed to solve puzzle in order to get stars.

Private int moveNr: Attribute to keep number of moves performed by player in order to solve particular puzzle.

Private const int moveLimit: Constant attribute to represent number of moves allowed to solve puzzle in order to get stars.

Methods:

Public Car getPlayersCar(): When invoked returns player's car (main car).

Public void resetLevel(): When invoked will reset number of moves and time for current puzzle.

Public void move(selectedCar Car, newPos Square[ ]): Method will call move() method of Car class to move selectedCar to new position.

Public Boolean puzzleSolved(): When invoked will call solved() method of puzzle to check if puzzle is solved.

Public int calculateStars(): When invoked will call updateAchievments() method of Puzzle class and retrieve number of stars.

Public int calculateCrowns(): When invoked will call updateAchievments() method of Puzzle class and retrieve number of crowns.

Public int calculateCoins(): When invoked will call updateCoins() method of Puzzle class and retrieve number of coins.

3.3.2 PUZZLE

FIGURE 23 PUZZLE

This class holds instance of GameGrid and will contain information about level number, puzzle number, crown number and whether puzzle is solved and also corresponding update methods.

Attributes:

private GameGrid grid: this attribute represents the game grid.

private int lvNr: attribute to hold number of level

private int puzzleNr: attribute to keep number indicating particular puzzle.

private Boolean solved: boolean to check whether puzzle is solved or not

private int starNr: attribute to keep number of stars achieved from solving current puzzle

private int crownNr: attribute to keep number of crowns achieved from solving current puzzle

Methods:

public int getLvNr(): get method to retrieve number of level

public void updateCoins(): when invoked calls **calculateCoins()** method from **Gamegrid** class to calculate number of coins and updates .txt file with new new number of coins.

public void updateAchievments(): when invoked calls **calculateCrowns()** and **calculateStars()** methods from **GameGrid** class to calculate number of crowns and stars to update them in .txt file accordingly.

public void opengrid(): when invoked initializes grid with appropriate puzzle.

3.3.3 SQUARE

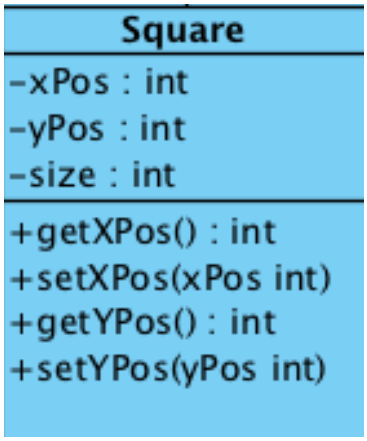

FIGURE 24 SQUARE

The class represents basic building block of the game. GameGrid and cars consist of array of squares.

Attributes:

private int xPos: attribute to keep position of X coordinate.

private int yPos: attribute to keep position of Y coordinate.

private int size: attribute to hold size of square.

Methods:

public int geXPos(): when method invoked retrieves position of X.

public int getYPos(): when method invoked retrieves position of Y.

public void setXPos(): method invoked to update position of X.

public void setYPos(): method invoked to update position of Y.

3.3.4 CAR

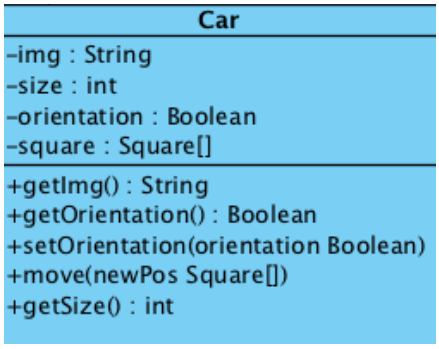

FIGURE 25 CAR

Instances of Car class will be made of several squares and the image of car model. Car objects will have methods to determine direction of movement.

Attributes:

private String img: Attribute to get image of model of car.

private int size: Attribute to keep size of car.

private Boolean orientation: Boolean to determine direction of movement for car. True will stand for vertical direction, False will stand for horizontal direction

private Square[] square: Attribute to keep array of squares of which car consists.

Methods:

public String getImg():

public Boolean getOrientation(): When invoked method returns orientation of car movement. True for vertical , False for horizontal.

public void setOrientation(): Method to set orientation for direction of car movement**.**

public void move (newPos Square [ ]): When invoked this method moves the car to a new position of adjacent squares.

public int getSize(): Method returns size of car.