BILKENT UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

CS319
Object Oriented Software Engineering

System Design Report
Project "Rush Hour"

Group 2.C Not So Oriented

Deniz Dalkılıç
21601896
Ahmet Ayrancıoğlu
21601206
Kaan Gönç
21602670
Ali Yümsel
21601841
Sina Şahan
21602609

Table Of Contents

1. Introduction

1.1 Purpose of the system

1.2 Design goals

2. High-level software architecture

2.1 Subsystem decomposition

2.2 Hardware/software mapping

2.3 Persistent data management

2.4 Access control and security

2.5 Boundary conditions

3. Low-level design

3.1 Object design trade-offs

3.2 Final object design

3.3 Packages

3.4 Class Interfaces

1. Introduction

1.1. Purpose of the System

Rush Hour is a sliding block logic game. It provides a fun way to improve one's problem solving, sequential thinking and logical reasoning skills by offering many different challenges. The purpose of our own system is to help our customers to improve their skills listed above by challenging themselves in an much more entertaining way. Our aim is to create a better Rush Hour gameplay experience in virtual life when compared to the real life, by changing the original Rush Hour game with our own additions of new features which we think will attract the customers a lot more. The players will have the opportunity to interact more with the game that they are not able to do such an action in real life except only moving real blocks. Players will feel more rewarded playing our game compared to the board game because of our level based reward system. Such a system will provide a better chance for the game to be served on different platforms with a great variety of features such as different theme or vehicle packs to different users according to their own interests which is also not possible in real life without paying an extra fee. Finally, adding new levels to the game and bringing the levels to the users will not require require users to buy new games and will be done with updates contrary to the board game.

1.2. Design Goals

Performance Criteria

Response Time

- The system should be able to define each input separately in the same frame when multiple inputs are given by the player at the same time and the system should respond to inputs from the user in less than 17ms.
- Progress saving should not take more than 0.5 seconds.

Throughput

- The system should be able to support smooth animations which means the game should be able to run at 60 frame per second.
- Transition between panels must take less than 1 second.

Memory

- Only needed space should be allocated in the system for the game to run.
- No unnecessary information about the game should be kept within the local directory.
- All of the progress details should be deleted when a user deletes his account as the data will be kept in a local directory.

Dependability Criteria

Reliability

- Any crashes during the gameplay should not effect the game data so that when a crash happens the player should be able to continue to his game from where he left.
- The system should automatically save the progress including the settings adjusted the player as a backup if the player quits the game without saving his progress.
- The system should be able to provide a local backup system so that the players do not worry about having an internet connection.

Robustness

- 99.9% of users who playthrough the game should not encounter a system crash that is caused by bugs. The system should not have any major bugs that can disrupt the overall gameplay experience of the user.
- The user should not be able to interact with any locked feature in the game which could create a corruption in the game data. In order to achieve such a system we will try every possible corner case that the users can encounter so that there won't be any problems revealed to the player.

Maintenance Criteria

Extendibility

- Game must be implemented with an extendible design, any additional classes created inside the Controller system should not affect the already existing behaviour of other controller classes and should start working as intended when the class is added to the Game Engine.
- Graphical updates or the complete change of the User Interface should be added without making any changes inside Model or Controller systems.

Modifiability

- Object oriented design concepts must be employed to make the game easily editable.
- Classes that depend on each other should not depend on the implementation of the methods used so that when changing the functionality of a method or adding new functionality to a method should not affect the classes that use that method.

Readability & Traceability

- The system should allow multiple people to work on the implementation at the same time by providing a maintainable structure in order to not have any difficulties in understanding the existing code segments, implementation of features and the design choices.

End User Criteria

Usability

- Players must be able to interact with the user interface without having any difficulty and without being given any extra information about how to use the interface. 95 out of 100 people should be able to start the game from the main menu when they open the game for the first time.
- User interface should use consistent background images for buttons and the colors used across menus should also be consistent.
- 95 out of 100 people, who never played rush hour before, should be able to understand and play the game after seeing the tutorial in the help menu. The tutorial should be made using images that describe and show how tha game is played instead of long textual descriptions.

Utility

- The players should be able to feel the metaphors of real world effectively such as hearing car engines and horns while playing a level.
- The system should inform the user on how effectively they solved the puzzles, in order to let the user learn more about development of their skills. The system should inform the user on how many moves the user made and the what the average number of moves made by players on that level is.

Cost Criteria

Development Cost

- The development should not take longer than 3 months.

Deployment Cost

- Cost of adding new features to the system should be low.

Maintenance Cost

- The cost for bug fixes and enhancements to the system should be low.

Trade-Offs (Technical, not Managerial)

Space vs. Speed

- If the system is not enough to meet the performance criteria such as response time or throughput requirements, we are planning to expand the memory allocation to speed up the system. However, we think there won’t be any problems with both the performance and the memory as our game will have an efficient design

Usability vs. Functionality

- Our design should not be complicated as it could decrease the usability of the product. Therefore, we won’t include any unnecessary function to our game and keep the basic functionalities so that the user can easily interact with the product without facing any difficulty in understanding the main use cases.

2. High-level Software Architecture

In our design, our plan is to divide our system into meaningful subsystems in order to deal with the problem of complexity and efficiency. Our main purpose is to reduce the amount of coupling between different systems such that a problem in a specific subsystem is less likely to affect another subsystem whereas keeping the cohesion between classes in a subsystem high. We are going to adjust the trade-off between cohesion and coupling in a meaningful way so that different individuals could easily develop different subsystems and combine them.

2.1. Subsystem Decomposition

In order to achieve our design goals we decided to use the architectural style Model/View/Controller (MVC) in our system. By using this design style we will maintain our application domain classes in the Model subsystems, user interface components in the View subsystems and the controller classes that manage the sequence of interactions with the user in the Controller subsystems. We are going to have a number of different subsystems for all of these three subsystem types of MVC style.

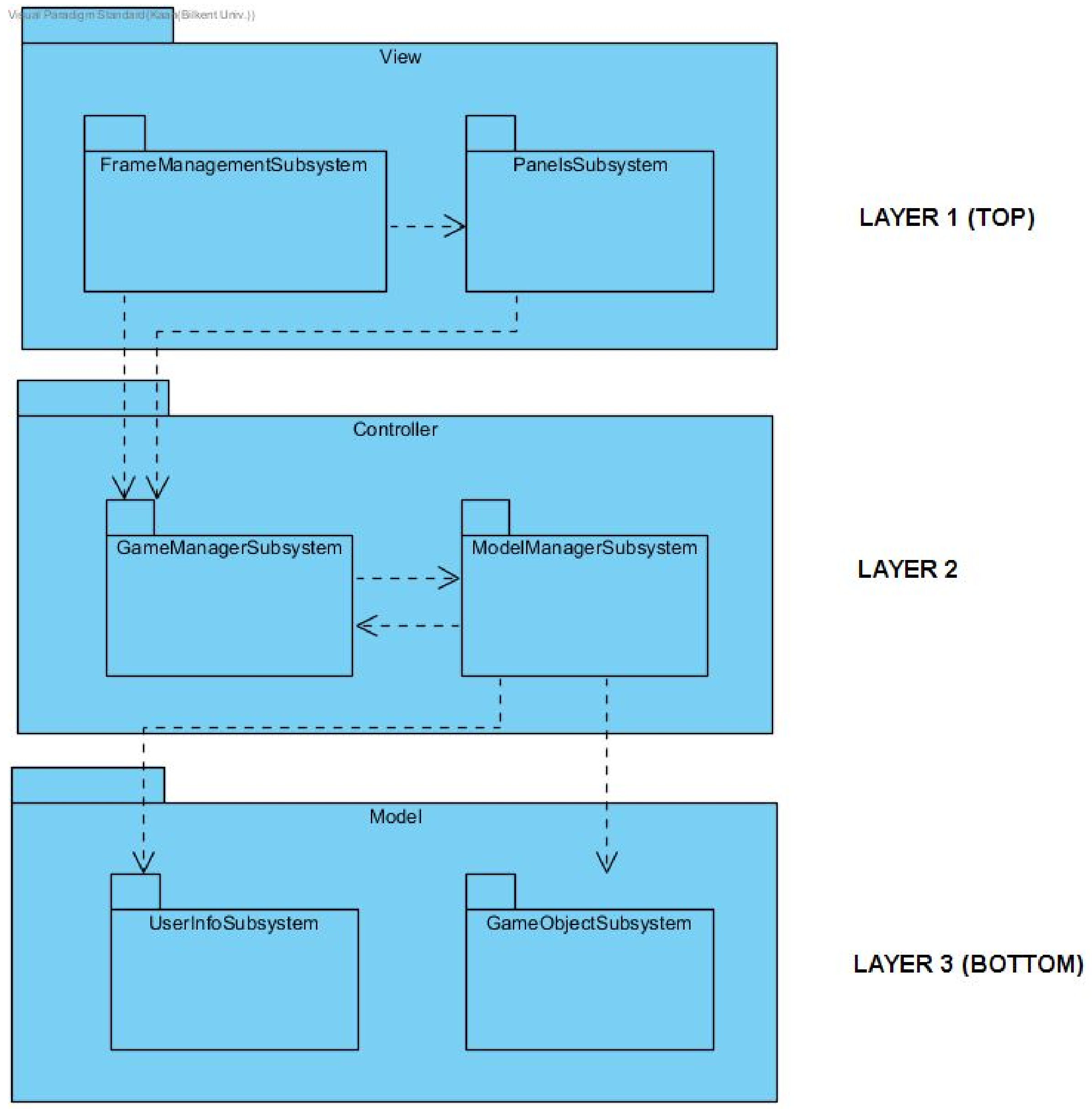

Figure 1 - Layered Subsystem Decomposition

.

As it is seen from the diagram, Layer 1 contains View subsystems, Layer 2 contains Controller subsystems, and Layer 3 contains Model subsystems. The purpose of layering is to show the abstract hierarchy between the subsystems. UserInfo and GameObject subsystems are the basic ones to declare the core objects of the game. ModelManager subsystem initiates the model objects, and make them work in a compatible way. GameManager is the main subsystem to make the game playable and user-interactable and provides the connection with the view. Panels subsystem designs and creates all the needed panels for the game. At last, FrameManagement subsystem gets data from the controller, shares them with the panels, and provides a reasonable harmony between the panels. More details about the components of these subsystems are shown in Figure 2.

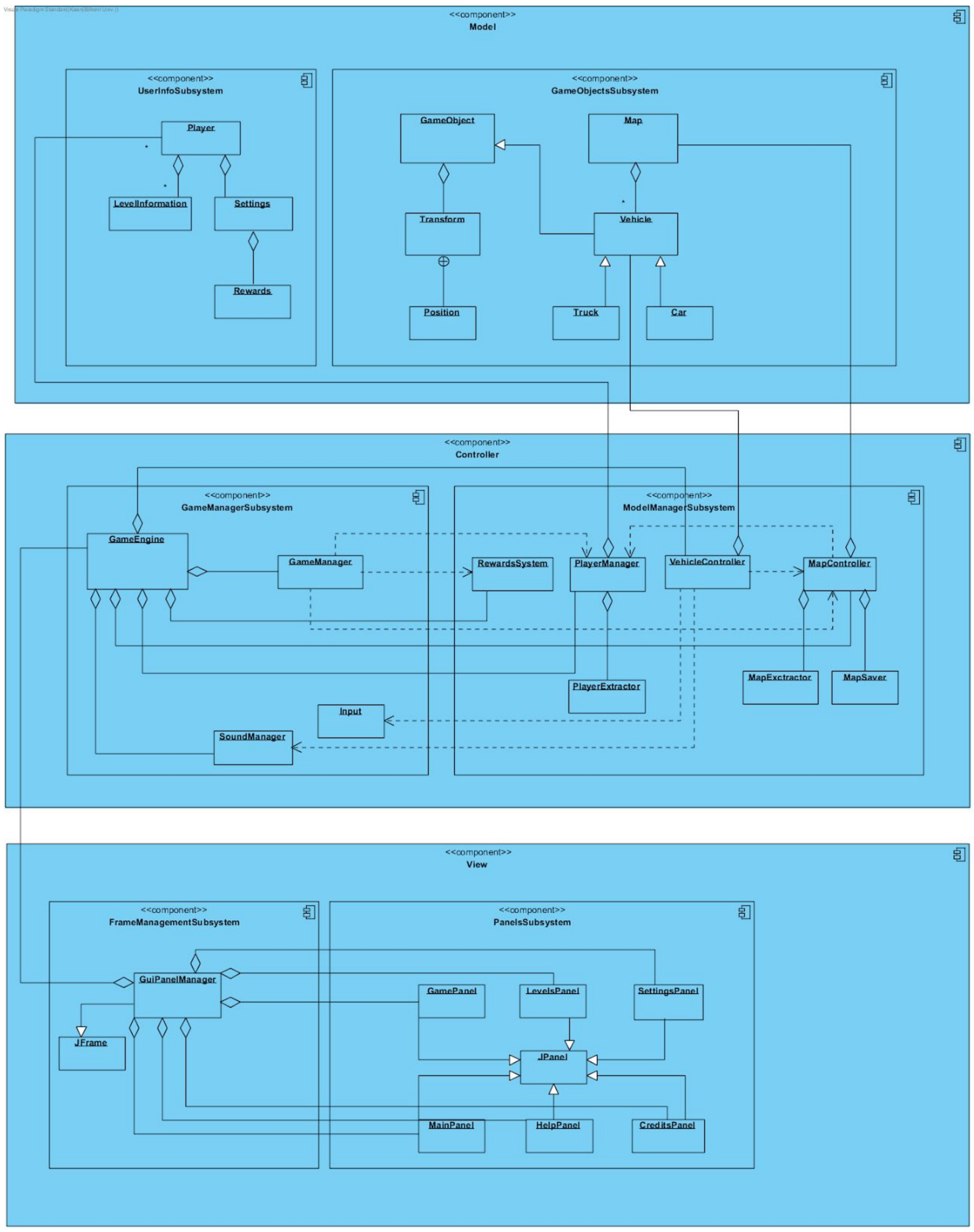

Figure 2 - Subsystems Component Diagram

2.2. Hardware / Software Mapping

The subsystems of Rush Hour will be implemented in Java programming language; more specifically for the development of the View subsystems we will refer to Java Swing libraries as well as improving the design with JavaFX libraries when necessary. Therefore, in order for users to execute the game, Java Runtime Environment (JRE) should be installed in their devices. In terms of the version of Java Development Kit (JDK) that the Rush Hour will require, we need to say that it will most probably require at least JDK8, as we will be doing our implementation with additions made from JavaFX libraries.

In terms of the hardware requirements of Rush Hour, the users will need a mouse(or a mouse pad) in order to interact with the panels in the game and also a keyboard in order to move the vehicles in the game scene. Our Input system, will be identifying such different inputs from the mouse and the keyboard so that the Controller subsystems will now which method to execute during gameplay. Moreover, our game won't be exporting or importing data from the internet so that the users should not worry about having an internet connection. The data will be tuned in to special format and will be kept in text files so that the saved game progresses will not cause a memory shortness in the system. Therefore, we are sure that any user who provides himself the requirements above will be able to play our game without having any performance issues.

2.3. Persistent Data Management

As Rush Hour is a relatively small Project, it does not require any complex data management system like a database. It will use the client's hard drive as storage and it will not require any internet connection after the initial installation of the game. All of the settings, maps, saved games and user data will be stored on the client's hard drive as text files. Saved games will have a metadata that stores which user owns this saved game and which level are they at; and it will have a map matrix which is the last state of the game map, just before the player exits. If the player selects "Play Game" option instead of choosing a level from levels menu, this file will be read and the player will start from that last state that he left the game. All of the textures and game object images will be stored in .png format as it allows transparent background (unlike .jpg) and it supports 24-bit and 48-bit color (unlike .gif which supports only 8-bit color). The sounds will be stored in .wav format.

2.4. Access control and security

As Rush Hour won't be connected to the internet and won't be using a database, there won't be much security issues that will affect the game. We will allow multiple users to play the game and have their own settings and saved games, however there will be no extra protection that inhibit a user to access another user's profile. As this game is stored locally on the client's hard drive, we assume that only trusted users

(users who have access to the computer) will have access to the game. Also, all of the user data will be machine specific (and all of the saved games will be user specific), so there won’t be a way to copy user data and saved games from another computer. Maps of the game will be stored as read-only, so that no third-party will be allowed to change the maps. However, in the later versions, we might add support to add new map-packs or user-created maps.

2.5. Boundary Conditions

2.5.1. Initialization

When RushHour is executed the very first time, the game will be initialized with a default player which does not have any progress with the game. However, when the game is executed at any other time, the game will be initialized according to the progress details of the last player that played the game. When a player interacts with a main menu option, current players progress details will be displayed in the associated panels. As, the progress details will be kept in local text files, the game will not have to communicate with another system to update the data during initialization, which significantly reduces the probability of a possible error, as no internet connection is needed with text file usage. Any data that is related to functionality of the game or the user will be kept in local directory as txt files inside the installation folder. However, the game will be opened from .exe file which the user can execute.

2.5.2. Termination

In order to terminate the game there are several options. Termination of the game can happen in two ways, the user can close the application by pressing the “X” button on the panel or the user can close the application by pressing the “Exit Game” button from the main menu. As we will have a dynamic progress saving system in our game (any new progress will directly update the data in the text files), the player that terminates the game by closing the application window will not lose any progress details. The system will update the data in case of a window termination with the “***windowClosing(WindowEvent e)”*** method of Swing library which is executed right before the window is closed.

2.5.3. Exception Handling

Rush Hour is an interactive game with images and many sound effects so it is can be hard to create a totally error-free system. Therefore, any audio or visual error that our implementation could not catch and fix may ruin the gameplay experience. So, we will provide a reload button for the players to reload the images and the sound system if they realize an error or misconfiguration during gameplay.

However, player’s progress will not be affected by such an reloading process as we will be using a dynamic progress saving system. The system will refresh the sound and images without making a change in the existing data.

3. Low-level design

3.1. Object design trade-offs

During the process of low level design we used some designs patterns which we thought would help us in developing the game. Each design pattern has its own tradeoff that we had to work around.

3.1.1. Singleton Design Pattern

To increase the maintainability of the game, we applied the singleton design pattern to our Manager and Controller classes which reside in Controller package. Our reasoning was, because there should only one manager or a controller that is actively controllering an aspect of the game. For example, there should only be 1 active “VehicleController” object that is controlling vehicles in the game. By using the singleton pattern we ensure that there can only be one instance of the class and other classes can easily access the instance of the class. However, using singleton design pattern has its advantages. As singleton object can be accessed easily by other classes and it is very convenient to use, it can be overused and lead to creating hidden dependencies between classes. Additionally, because the reference is not being actively stored in the classes that use the singleton class, tracking the dependencies becomes difficult. This leads to increased coupling between our Manager classes. Finally, using singletons can cause to problems when writing testable codes, because of the dependencies and the coupling. So for testing writing almost fully functional classes becomes necessary which is time consuming.

3.1.2. Model View Controller (MVC) Design Pattern

We wanted to use iterative engineering while developing our games to easily design, implement, test and change our implementations as necessary. By using MVC design pattern we ensured that multiple people could design and develop different aspects of our game such as Model, View and the Controller as the name of the pattern implies. This allowed us to divide the workflow between group members more effectively and lead to a faster design and development time. Another benefit of using MVC was while testing the game as we could use multiple views while using the MVC pattern and we did not have to create a fully functional user interface to make sure our game logic was working as intended. However, designing our game with MVC design pattern took longer than if we have used other patterns because deciding how to divide needed classes into three concrete parts and making sure that coupling between classes especially in different packages would be low was a time consuming process. Another, disadvantage of MVC is View depends highly on both on Controller and the Model, so to create and test a functioning user interface, we had to have a functioning Model and Controller.

3.1.3. Composite Design Pattern

We wanted all our controller and manager classes to extend from a single base class simply called “Manager”. Our GameEngine class initializes and holds every single one of our classes that are derived from “Manager” class in a Manager array. This simplifies the GameEngine class and unifies our Controller subsystem. The Manager class has an empty update() method which can be overwritten by each child class to fit their own functionalities. In our GameEngine class, update methods of all our Manager classes is called 60 times every second (60fps) regardless of their implementations. This allows us to change, add or remove any functionality of an existing classe’s update method without worrying about altering any other class. Another benefit of using a Manager base class is adding new classes, which derive from the Manager class, becomes very easily as we only have to add the object of the newly created class to the GameEngine without changing any code segments in our GameEngine. After that we only need to worry about the functionality of the class we added as any logic that is related to that class is contained in that class. However, using composite design patterns has its own drawbacks, It creates a too general class design that some of our Manager classes does not need to derive from but still derive from to maintain uniformity.

3.2. Final Object Design

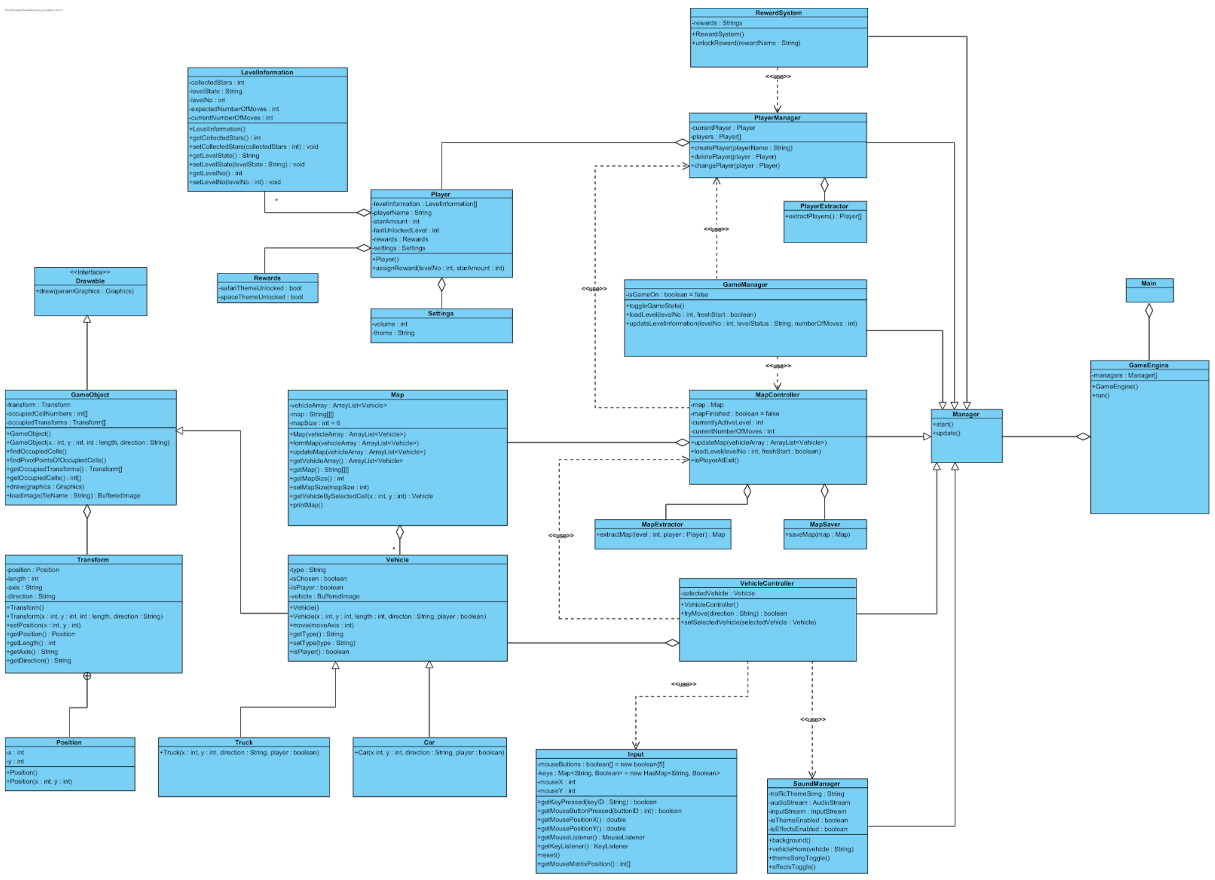
RewardSystem
-rewards : Strings
+RewardSystem()
+unlockReward(rewardName : String)
<<use>>
PlayerManager
-currentPlayer : Player
-players : Player[]
+createPlayer(playerName : String)
+deletePlayer(player : Player)
+changePlayer(player : Player)
PlayerExtractor
+extractPlayers() : Player[]
LevelInformation
-collectedStars : int
-levelState : String
-levelNo : int
-expectedNumberOfMoves : int
-currentNumberOfMoves : int
+LevelInformation()
+getCollectedStars() : int
+setCollectedStars(collectedStars : int) : void
+getLevelState() : String
+setLevelState(levelState : String) : void
+getLevelNo() : int
+setLevelNo(levelNo : int) : void
*
Player
-levelInformation : LevelInformation[]
-playerName : String
-starAmount : int
-lastUnlockedLevel : int
-rewards : Rewards
-settings : Settings
+Player()
+assignReward(levelNo : int, starAmount : int)
Rewards
-safariThemeUnlocked : bool
-spaceThemeUnlocked : bool
Settings
-volume : int
-theme : String
<<Interface>>
Drawable
+draw(paramGraphics : Graphics)
<<use>>
<<use>>
GameManager
-isGameOn : boolean = false
+toggleGameState()
+loadLevel(levelNo : int, freshStart : boolean)
+updateLevelInformation(levelNo : int, levelStatus : String, numberOfMoves : int)
<<use>>
Main
GameEngine
-managers : Manager[]
+GameEngine()
+run()
Manager
+start()
+update()
GameObject
-transform : Transform
-occupiedCellNumbers : int[]
-occupiedTransforms : Transform[]
+GameObject()
+GameObject(x : int, y : int, int : length, direction : String)
+findOccupiedCells()
+findPivotPointsOfOccupiedCells()
+getOccupiedTransforms() : Transform[]
+getOccupiedCells() : int[]
+draw(graphics : Graphics)
+loadImage(fileName : String) : BufferedImage
Map
-vehicleArray : ArrayList<Vehicle>
-map : String[][]
-mapSize : int = 6
+Map(vehicleArray : ArrayList<Vehicle>)
+formMap(vehicleArray : ArrayList<Vehicle>)
+updateMap(vehicleArray : ArrayList<Vehicle>)
+getVehicleArray() : ArrayList<Vehicle>
+getMap() : String[][]
+getMapSize() : int
+setMapSize(mapSize : int)
+getVehicleBySelectedCell(x : int, y : int) : Vehicle
+printMap()
MapController
-map : Map
-mapFinished : boolean = false
-currentlyActiveLevel : int
-currentNumberOfMoves : int
+updateMap(vehicleArray : ArrayList<Vehicle>)
+loadLevel(levelNo : int, freshStart : boolean)
+isPlayerAtExit()
MapExtractor
+extractMap(level : int, player : Player) : Map
MapSaver
+saveMap(map : Map)
<<use>>
*
Transform
-position : Position
-length : int
-axis : String
-direction : String
+Transform()
+Transform(x : int, y : int, int : length, direction : String)
+setPosition(x : int, y : int)
+getPosition() : Position
+getLength() : int
+getAxis() : String
+getDirection() : String
Vehicle
-type : String
-isChosen : boolean
-isPlayer : boolean
-vehicle : BufferedImage
+Vehicle()
+Vehicle(x : int, y : int, length : int, directon : String, player : boolean)
+move(moveAxis : int)
+getType() : String
+setType(type : String)
+isPlayer() : boolean
VehicleController
-selectedVehicle : Vehicle
+VehicleController()
+tryMove(direction : String) : boolean
+setSelectedVehicle(selectedVehicle : Vehicle)
<<use>>
<<use>>
Position
-x : int
-y : int
+Position()
+Position(x : int, y : int)
Truck
+Truck(x : int, y : int, direction : String, player : boolean)
Car
+Car(x int, y : int, direction : String, player : boolean)
Input
-mouseButtons : boolean[] = new boolean[5]
-keys : Map<String, Boolean> = new HasMap<String, Boolean>
-mouseX : int
-mouseY : int
+getKeyPressed(keyID : String) : boolean
+getMouseButtonPressed(buttonID : int) : boolean
+getMousePositionX() : double
+getMousePositionY() : double
+getMouseListener() : MouseListener
+getKeyListener() : KeyListener
+reset()
+getMouseMatrixPosition() : int[]
SoundManager
-trafficThemeSong : String
-audioStream : AudioStream
-inputStream : InputStream
-isThemeEnabled : boolean
-isEffectsEnabled : boolean
+background()
+vehicleHorn(vehicle : String)
+themeSongToggle()
+effectsToggle()

Figure 3 - Complete Class Diagram (Model - Controller)

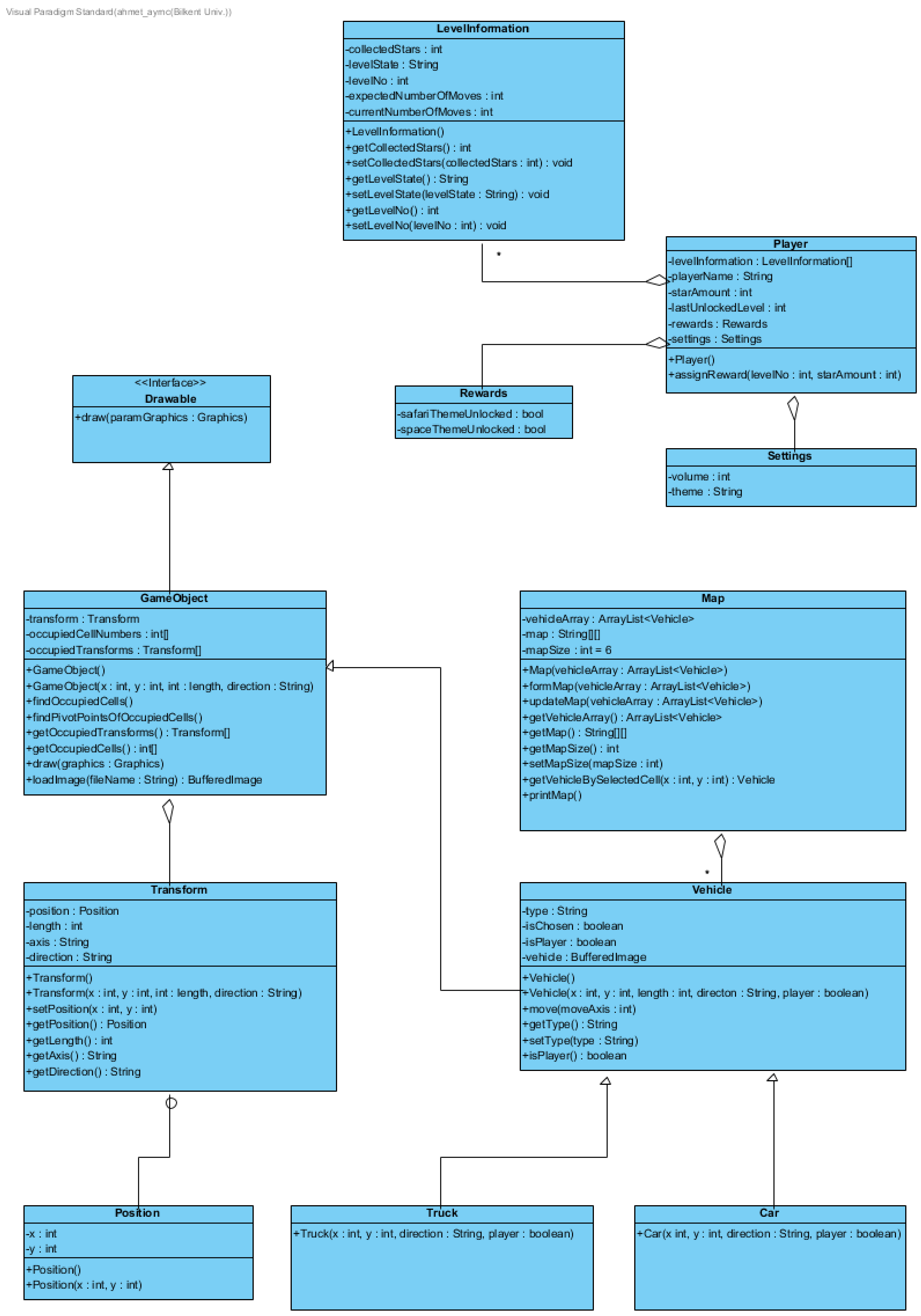
Visual Paradigm Standard(ahmet_aymc(Bilkent Univ.))
LevelInformation
-collectedStars : int
-levelState : String
-levelNo : int
-expectedNumberOfMoves : int
-currentNumberOfMoves : int
+LevelInformation()
+getCollectedStars() : int
+setCollectedStars(collectedStars : int) : void
+getLevelState() : String
+setLevelState(levelState : String) : void
+getLevelNo() : int
+setLevelNo(levelNo : int) : void
*
Player
-levelInformation : LevelInformation[]
-playerName : String
-starAmount : int
-lastUnlockedLevel : int
-rewards : Rewards
-settings : Settings
+Player()
+assignReward(levelNo : int, starAmount : int)
Rewards
-safariThemeUnlocked : bool
-spaceThemeUnlocked : bool
Settings
-volume : int
-theme : String
<<Interface>>
Drawable
+draw(paramGraphics : Graphics)
GameObject
-transform : Transform
-occupiedCellNumbers : int[]
-occupiedTransforms : Transform[]
+GameObject()
+GameObject(x : int, y : int, int : length, direction : String)
+findOccupiedCells()
+findPivotPointsOfOccupiedCells()
+getOccupiedTransforms() : Transform[]
+getOccupiedCells() : int[]
+draw(graphics : Graphics)
+loadImage(fileName : String) : BufferedImage
Map
-vehicleArray : ArrayList<Vehicle>
-map : String[][]
-mapSize : int = 6
+Map(vehicleArray : ArrayList<Vehicle>)
+formMap(vehicleArray : ArrayList<Vehicle>)
+updateMap(vehicleArray : ArrayList<Vehicle>)
+getVehicleArray() : ArrayList<Vehicle>
+getMap() : String[][]
+getMapSize() : int
+setMapSize(mapSize : int)
+getVehicleBySelectedCell(x : int, y : int) : Vehicle
+printMap()
*
Transform
-position : Position
-length : int
-axis : String
-direction : String
+Transform()
+Transform(x : int, y : int, int : length, direction : String)
+setPosition(x : int, y : int)
+getPosition() : Position
+getLength() : int
+getAxis() : String
+getDirection() : String
Vehicle
-type : String
-isChosen : boolean
-isPlayer : boolean
-vehicle : BufferedImage
+Vehicle()
+Vehicle(x : int, y : int, length : int, directon : String, player : boolean)
+move(moveAxis : int)
+getType() : String
+setType(type : String)
+isPlayer() : boolean
Position
-x : int
-y : int
+Position()
+Position(x : int, y : int)
Truck
+Truck(x : int, y : int, direction : String, player : boolean)
Car
+Car(x int, y : int, direction : String, player : boolean)

Figure 4 - Class Diagram (Model)

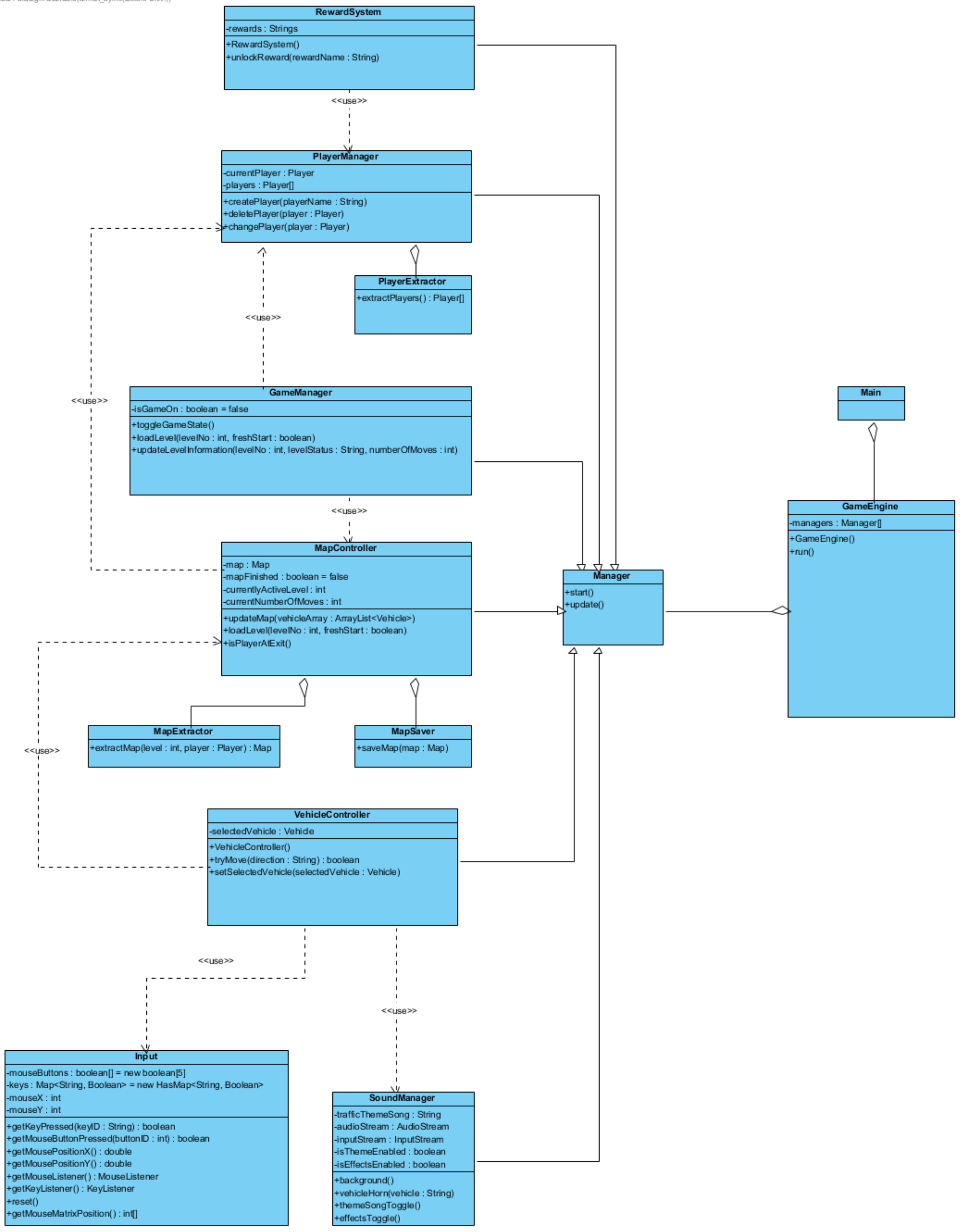
RewardSystem
-rewards : Strings
+RewardSystem()
+unlockReward(rewardName : String)
<<use>>
PlayerManager
-currentPlayer : Player
-players : Player[]
+createPlayer(playerName : String)
+deletePlayer(player : Player)
+changePlayer(player : Player)
PlayerExtractor
+extractPlayers() : Player[]
<<use>>
<<use>>
GameManager
-isGameOn : boolean = false
+toggleGameState()
+loadLevel(levelNo : int, freshStart : boolean)
+updateLevelInformation(levelNo : int, levelStatus : String, numberOfMoves : int)
<<use>>
MapController
-map : Map
-mapFinished : boolean = false
-currentlyActiveLevel : int
-currentNumberOfMoves : int
+updateMap(vehicleArray : ArrayList<Vehicle>)
+loadLevel(levelNo : int, freshStart : boolean)
+isPlayerAtExit()
MapExtractor
+extractMap(level : int, player : Player) : Map
MapSaver
+saveMap(map : Map)
<<use>>
VehicleController
-selectedVehicle : Vehicle
+VehicleController()
+tryMove(direction : String) : boolean
+setSelectedVehicle(selectedVehicle : Vehicle)
<<use>>
<<use>>
Input
-mouseButtons : boolean[] = new boolean[5]
-keys : Map<String, Boolean> = new HasMap<String, Boolean>
-mouseX : int
-mouseY : int
+getKeyPressed(keyID : String) : boolean
+getMouseButtonPressed(buttonID : int) : boolean
+getMousePositionX() : double
+getMousePositionY() : double
+getMouseListener() : MouseListener
+getKeyListener() : KeyListener
+reset()
+getMouseMatrixPosition() : int[]
SoundManager
-trafficThemeSong : String
-audioStream : AudioStream
-inputStream : InputStream
-isThemeEnabled : boolean
-isEffectsEnabled : boolean
+background()
+vehicleHorn(vehicle : String)
+themeSongToggle()
+effectsToggle()
Manager
+start()
+update()
Main
GameEngine
-managers : Manager[]
+GameEngine()
+run()

Figure 5 - Class Diagram (Controller)

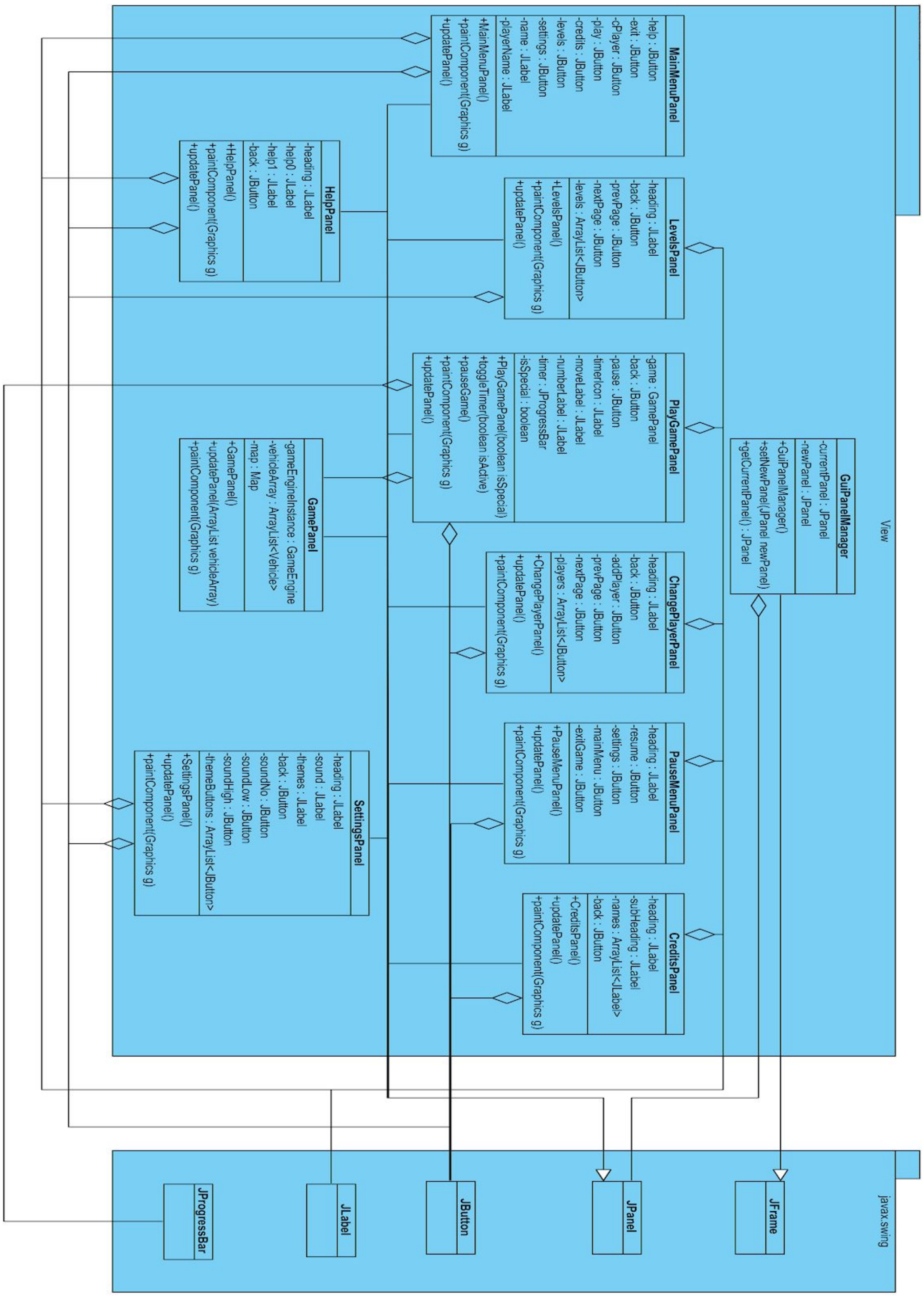

Figure 6 - Class Diagram (View)

GamePanel also aggregates GameEngine, Vehicle and Map, but those are omitted for readability.

3.3. Packages

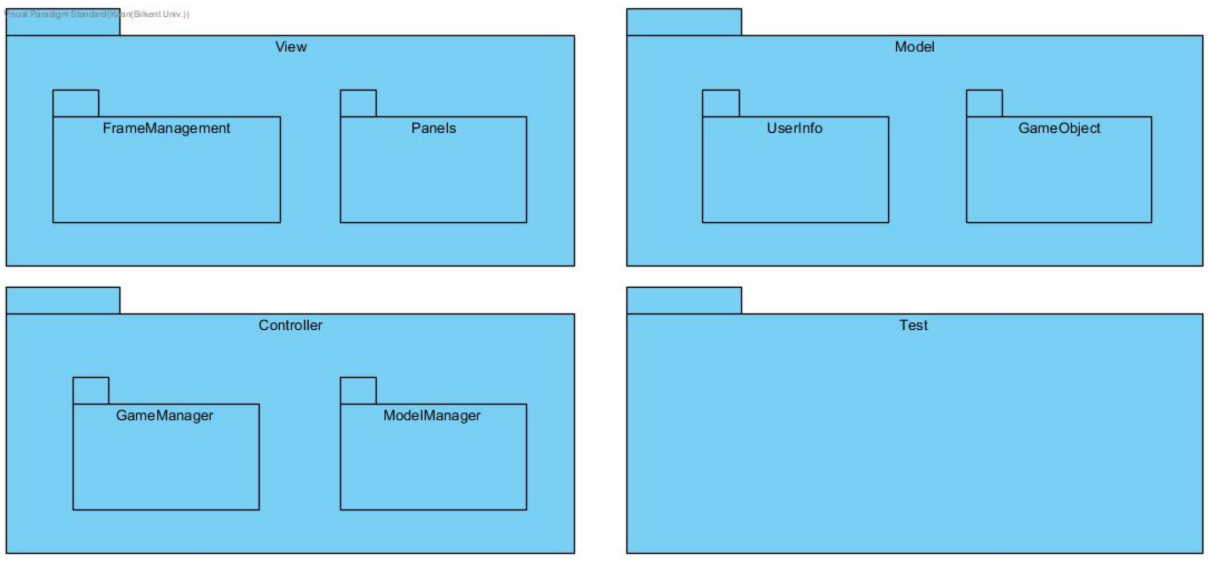

Figure 7 - Package Diagram

Model, View, and Controller packages and the packages inside them represent their subsystem versions as shown in Figure 1. Additionally, there is Test package that contains the test classes in order to test the game via the Java console rather than graphical user interfaces.

3.4. Class Interfaces

3.4.1. Position Class

Position class is used to hold the coordinates of a Transform in two dimensional space.

Attributes

- **int x :** Represents the position on the x axis in two dimensional space.
- **int y:** Represents the position on the x axis in two dimensional space.

Constructors

- **public Position() :** Empty constructor that initializes both values to 0.
- **public Position(int x, int y) :** Initializes the position with the given x and y values on the two dimensional space.

3.4.2. Transform Class

Transform class is used to hold every possible information that every 2 dimensional object could have. We use position, length and direction to represent the 2d object's position, rotation and scale respectively.

Attributes

- **Position position :** Represents the position of the object in 2d space.
- **int length:** Represents the length of the object.
- **String axis:** Represents the axis of the object, horizontal or vertical.
- **String direction:** Represents the direction of the object, right, left, up or down.

Constructors

- **public Transform() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public Position getPosition():** Returns the position as a Position.
- **public void setPosition(int x, int y):** Sets the position attribute with the given x and y values.
- **public int getLength():** Returns the length as int.
- **public String getAxis():** Returns the axis as String.
- **public String getDirection():** Returns the direction as String.

3.4.3. GameObject Class

GameObject class is used to represent every possible object that is inside the game.

Attributes

- **Transform transform :** Represents all attributes that is related to 2d space.
- **int[] occupiedCellNumbers:** Holds all the cells that this game object currently occupies.
- **Transform[] occupiedTransforms:** Represents all other transforms that is related to this game object.

Constructors

- **public GameObject() :** Empty constructor that initializes values to their specified initial values.

Methods

- **private void findOccupiedCells():** Updates the cells that is occupied by this game object.
- **private void findPivotPointsOfOccupiedCells():** Updates the pivots of the occupied cells.
- **public Transform[] getOccupiedTransforms():** Returns the occupied Transforms.
- **public int[] getOccupiedCells():** Returns the occupied cells.
- **public void draw():** Draws the image connected to this game object.
- **private void loadImage:** Loads the image connected to this game object from local directory.

3.4.4. Vehicle Class

Vehicle class is used to represent the vehicles in our game. They are the main game object in the game Rush Hour.

Attributes

- **Boolean isPlayer:** To determine whether this vehicle is the playerCar.
- **BufferedImage vehicleImage:** Holds the image connected to this vehicle.

Constructors

- **public Vehicle() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void move(int moveAxis):** Moves the amount that is given according to its axis inside the Transform property. Moves backwards when the value is negative.
- **public boolean isPlayer():** Returns whether this vehicle is the player vehicle.

3.4.5. Map Class

Map class is used to represent the map that the game is played on. It holds the vehicles that are part of the level.

Attributes

- **ArrayList<Vehicle> vehicleArray:** Hold the vehicle that are currently part of the level.
- **String[] map:** Representation of the map in textual form.

- **int mapSize:** Represents the size of the map. A size of 6 means that the map is 6 by 6.

Constructors

- **public Map() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void formMap(ArrayList<Vehicle> vehicleArray):** Forms the textual form of the map from the given array list of vehicles.
- **public void updateMap(ArrayList<Vehicle> vehicleArray):** Updates the textual form of the map from the given array list of vehicles.
- **public ArrayList<Vehicle> getVehicleArray():** Returns the vehicles on the map as ArrayList.
- **public String[][] getMap():** Returns the textual representation of the map as 2d String array.
- **public int getMapSize():** Returns the size of the map.
- **public Vehicle getVehicleBySelectedCell(int x, int y):** Returns the vehicle occupying the position given as parameter.
- **public void printMap(): prints the map in a textual form to the console.**

3.4.6. Drawable Interface

An interface for GameObjects which are drawable.

Methods

- **public void draw(Graphics graphics):** Represents the draw method that is going to be implemented by other classes.

3.4.7. LevelInformation Class

LevelInformation class holds all information about a particular level such as the level no, expected number of moves and the amount of stars collected on that level.

Attributes

- **int CollectedStars:** Represents the stars collected by the player in that level.
- **String levelState:** Represents the state of the level. For example: Completed, NeverPlayed and such.

- **int levelNo:** Represents the level with a number.
- **int expectedNumberOfMoves:** Holds the number of moves to reach to the exit.
- **nt currentNumberOfMoves:** If the level is not completed and is in a saved state, hold the current number of moves the user has made in that level.

Constructors

- **public LevelInformation() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public int getCollectedStars():** Returns the number of sars as int.
- **public void setCollectedSars(int collectedStars):** Sets the collected stars on this particular level, checks if the new value is higher than the old value and does not update the value if it is lower.
- **public String getLevelState():** Returns the state of the level as String.
- **public void setLevelState(String levelState):** Changes the level state to the given value.
- **public int getLevelNo():** Returns the level number.

3.4.8. Rewards Class

Rewards class is used to hold the reward information of a player.

Attributes

- **boolean spaceThemeUnlocked:** Shows whether the Space Theme is unlocked or not.
- **boolean safariThemeUnlocked:** Shows whether the Safari Theme is unlocked or not.

3.4.9. Settings Class

Settings class is used to hold settings information of a player

Attributes

- **int volume:** Represents the selected volume of the player.
- **String theme:** Represents the selected theme by the player.

3.4.10. Player Class

Player class hold represent the Player who is playing the game. It holds every information about a player such as, level progression, rewards and settings.

Attributes

- **LevelInformation[] levelInformations:** Holds the information about every level spesific to the user.
- **String playerName:** Represent the name of the player.
- **int starAmount:** Represents the total number of stars collected from every level.
- **int lastUnlockedLevel:** Holds the number of the last unlocked level by this player.
- **Rewards rewards:** Holds all information about the players rewards.
- **Settings settings**: Holds all information about the players settings.

Constructors

- **public Player() :** Empty constructor that initializes values to their specified initial values.

3.4.11. RewardSystem Class

RewardsSystem class is responsible for unlocking rewards for the users.

Attributes

- **String[] rewards:** Represents all possible rewards that is available in the game.

Constructors

- **public RewardSystem() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void unlockReward(String rewardName):** Unlocks the reward specified by the parameter for the currently active user.

3.4.12. PlayerManager Class

PlayerManager is responsible for handling the updates of player information, creating, changing and deleting users.

Attributes

- **Player currentPlayer:** Represents the currently active user,
- **Player[] players:** Holds every player that is created for the game.

Constructors

- **public PlayerManager() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void createPlayer(String playerName):** Creates a new fresh player with the given name and adds the player to the players list.
- **public void deletePlayer(Player player):** Deletes all the data of the given user and removes it from the players list.
- **public void changePlayer(Player player):** Changes the currently active player to the given Player.

3.4.13. PlayerExtractor Class

PlayerExtractor is responsible for extracting the player data from the local directory in to the game.

Methods

- **public Player[] extractPlayers():** Extracts the player information from local directory, creates a Player array to save the extracted information and return the Player array.

3.4.14. GameManager Class

GameManager is a generic class that handles higher aspects of the game.

Attributes

- **boolean isGameOn:** Shows whether the game is currently on or the game is not on like paused game or menus.

Constructors

- **public GameManager() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void toggleGameState():** Toggles the game state between on and off.
- **public void loadLevel(int levelNo, boolean freshStart):** Load the level given the level no. If the fresh start is true, loads the game in a fresh state, if it is false and the game has a already saved state, loads the level from the saved state.
- **public void updateLevelInformation(int levelNo, String, levelStatus, int numberOfMoves):** updates the level nformation of the given level with the information that is passes as a parameter.

3.4.15. MapController Class

MapController class is responsible for loading, saving levels. Updating the current map and holding information about the level.

Attributes

- **Map map:** Represents the map that the level is played on.
- **boolean levelFinished:** Shows whether the level is finished or being played.
- **int currentlyActiveLevel:** Stores the level no of the current level.
- **int currentNumberOfMoves:** Stores the current number of number that the player made in this active level.

Constructors

- **public MapController() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void updateMap(ArrayList<Vehicle> vehicleArray):** Updates the map object with the given vehicle arrayList.
- **public void loadLevel(int levelNo, boolean freshStart):** Load the level given the level no. If the fresh start is true, loads the game in a fresh state, if it is false and the game has a already saved state, loads the level from the saved state.
- **public boolean isPlayerAtExit():** Returns true if the player is at the exit.

3.4.16. MapExtractor Class

MapExtractor is responsible for extracting the map data from the local directory into the game.

Methods

- **public Player[] extractMap(int levelNo, Player player):** Extracts the map of the given level from the directory of the given player.

3.4.17. MapSaver Class

MapSaver is responsible for saving the map data from the game into the local directory.

Methods

- **public void saveMap(Map map):** Saves the map in a specified format to the local directory.

3.4.18. VehicleController Class

VehicleController class is responsible for controlling the vehicles on the map.

Attributes

- **Vehicle selectedVehicle:** Represents the currently selected vehicle by the user.

Constructors

- **public VehicleController() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public boolean tryMove(String direction):** Tries to move the vehicle 1 cell in the direction given and tests whether the move made by player is a valid move. If the vehicle can move in the given direction moves the vehicle and returns true, else it does not move the vehicle and returns false.

- **public void setSelectedVehicle(Vehicle selectedVehicle):** Sets the selected vehicle attribute to the given Vehicle.

3.4.19. SoundManager Class

SoundManager class is responsible for all the sounds that can be heard in the game.

Attributes

- **String trafficThemeSong:** Holds the directory of the sound effect.
- **AudioStream audioStream:**
- **InputStream inputStream:**
- **boolean isThemeEnabled:** Shows whether a theme is enabled or not.
- **boolean isEffectsEnabled:** Shows whether the sound effects for this player is enabled or not.

Constructors

- **public SoundManager() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void background():** Plays the background music.
- **public void vehicleHorn():** Plays the vehicle horn sound effect.
- **public void themeSongToggle():** Toggles the state of the theme song, on or off.
- **public void effectsToggle():** Toggles the state of the sound effects, on or off.

3.4.20. Input Class

Input class is responsible for handling player inputs.

Attributes

- **boolean[] mouseButtons:** Holds booleans that represents 5 mouse buttons, whether they are pressed or not.
- **Map<String, Boolean> keys:** Holds the booleans for keys, whether they are pressed or not and connects the booleans and the strings that represent keys.
- **int mouseX:** Holds the x position of the mouse in terms of pixels.
- **int mouseY:** Holds the y position of the mouse in terms of pixels.

Constructors

- **public Input() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public boolean getKeyPressed(String keyID):** Returns true during the frame the user presses the the key given as parameter.
- **public boolean getMouseButtonPressed(int buttonID):** Returns true during the frame the user presses the the mouse button given as parameter.
- **public int getMousePositionX():** Returns the current mouse position’s x value in terms of pixels.
- **public int getMousePositionY():** Returns the current mouse position’s x value in terms of pixels.
- **public int[] getMouseMatrixPosition():** Returns the current mouse position in term of cells.
- **public MouseListener getMouseListener():** Returns the mouse listener used by the input class.
- **public KeyboardListener getKeyboardListener():** Returns the keyboard listener used by the input class.

3.4.21. Manager Class

Manager class is the base class that every manager or controller is derived from.

Methods

- **public void start():** Called when the object is created before update methods. It is only called once.
- **public void update():** Called every frame from the Game Engine.

3.4.22. GameEngine Class

GameEngine class is responsible for handling the game loop and every class that is related to the game logic.

Attributes

- **Manager[] managers:** Holds every manager that is being used to run the game.

Constructors

- **public GameEngine() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void run():** Calls the update methods of every Manager object that is inside managers array.

3.4.23. GuiPanelManager Class

GuiPanelManager class extends javax.swing.JFrame class, and it is responsible for managing the different panels of the game.

Attributes

- **JPanel currentPanel :** The panel that is currently displayed on the screen.
- **JPanel newPanel :** The panel that will be displayed next. This will be different from current panel only when a panel changing action is done but the next panel is not ready to be displayed.

Constructors

- **public GuiPanelManager() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void setNewPanel(JPanel newPanel) :** Makes the newPanel ready to display, assigns the currentPanel with newPanel and sets the currentPanel visible.
- **public JPanel getCurrentPanel():** Returns the current panel to the caller.

3.4.24. MainMenuPanel Class

The panel that holds the Main Menu Layout for our game

Attributes

- **JButton help :** The help button. When pressed, the HelpPanel is displayed.
- **JButton exit :** The exit button. When pressed, the game checks if there is any active file reads and writes and when it is finished closes the game.
- **JButton cPlayer :** The change player button. When pressed, the ChangePlayerPanel is displayed.
- **JButton play :** The play game button. When pressed, the PlayGamePanel is displayed.

- **JButton credits :** The credits button. When pressed, the CreditsPanel is displayed.
- **JButton levels :** The levels button. When pressed, the LevelsPanel is displayed.
- **JButton settings:** The settings button. When pressed, the SettingsPanel is displayed.
- **JLabel name:** The label that displays the name of the game (Rush Hour).
- **JLabel playerName:** The label that displays the name of the current player. It also holds a small icon next to the player name.

Constructors

- **public MainMenuPanel() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.
- **public void updatePanel():** The method that updates the panel to display the changes on the screen.

3.4.25. LevelsPanel Class

The panel that holds the Levels Menu Layout for our game

Attributes

- **JButton prevPage :** The previous page button. When pressed, the previous page of level buttons is displayed (if the previous page exists).
- **JButton nextPage :** The next page button. When pressed, the next page of level buttons is displayed (if the next page exists).
- **JButton back :** The back button. When pressed, the previous panel is displayed (Main Menu).
- **ArrayList<JButton> levels :** The list of buttons that represents the different levels of the game. When pressed PlayGamePanel is displayed with the correct level loaded
- **JLabel heading:** The label that displays the heading to this panel (Levels)

Constructors

- **public LevelsPanel() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.

- **public void updatePanel():** The method that updates the panel to display the changes on the screen.

3.4.26. HelpPanel Class

The panel that holds the Help Screen for our game

Attributes

- **JButton back :** The back button. When pressed, the previous panel is displayed (Main Menu).
- **JLabel heading:** The label that displays the heading to this panel (How To Play).
- **JLabel help0:** The first help that is displayed (on the left side). It contains an image and a text explaining how to play.
- **JLabel help1:** The second help that is displayed (on the right side). It also contains an image and a text explaining how to play.

Constructors

- **public HelpPanel() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.
- **public void updatePanel():** The method that updates the panel to display the changes on the screen.

3.4.27. PlayGamePanel Class

The panel that holds the Layout for the Play Game Screen. The actual gameplay is in the GamePanel panel which is held inside this panel as the game attribute.

Attributes

- **JButton back :** The help button. When pressed, the previous panel is displayed (Main Menu or Levels). The current state of the game is also saved when this button gets pressed.
- **JButton pause :** The pause button. When pressed, the PauseMenuPanel is displayed.
- **JLabel TimerIcon :** The label that holds the timer icon.
- **JLabel moveLabel :** The label that displays the message “Number of Moves”.

- **JLabel numberLabel :** The label that displays the current number of moves that the user made.
- **JProgressBar timer :** The progress bar that shows how much time left in a special level. This timer is disabled when the level is not a special level.
- **GamePanel game :** The GamePanel instance that allows to play current level.
- **boolean isSpecial :** The boolean that holds the value of whether the level is special or not. This is needed to toggle time on or off.

Constructors

- **public PlayGamePanel(boolean isSpecial) :** Class constructor that initializes values to their specified initial values. isSpecial parameter decides whether the current level is special or not.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.
- **public void updatePanel():** The method that updates the panel to display the changes on the screen.
- **public void toggleTimer(boolean isActive):** Toggles the timer to be active or disabled.
- **public void pauseGame():** Puts a 50% transparent screen on the panel before the PauseMenuPanel is displayed to make PauseMenuPanel appear more distinct from the previous panel (as the PauseMenuPanel will not cover the whole screen).

3.4.28. GamePanel Class

The panel that the actual game is running.

Attributes

- **GameEngine gameEngineInstance :** The current instance of the game engine. This allows easy access to the data like the current map.
- **ArrayList<Vehicle> vehicleArray :** The array that holds the vehicles on the map.
- **Map map :** The current map to display. Every time that the map changes, this changes as well to display the vehicle positions correctly

Constructors

- **public GamePanel() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.
- **public void updatePanel(ArrayList<Vehicle> vehicleArray):** The method that updates the panel to display the changes on the screen. This one takes the current vehicleArray every time that a move happens in the game.

3.4.29. ChangePlayerPanel Class

The panel that holds the Levels Menu Layout for our game

Attributes

- **JButton back :** The back button. When pressed, the previous panel is displayed (Main Menu).
- **JButton addPlayer :** The add player button. When pressed, a new player is created and added to the player list.
- **JButton prevPage :** The previous page button. When pressed, the previous page of players is displayed (if the previous page exists).
- **JButton nextPage :** The next page button. When pressed, the next page of players is displayed (if the next page exists).
- **ArrayList<JButton> players :** The list that holds all the player buttons.
- **JLabel heading:** The label that displays the heading to this panel (Change Player).

Constructors

- **public ChangePlayerPanel() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.
- **public void updatePanel():** The method that updates the panel to display the changes on the screen.

3.4.30. PauseMenuPanel Class

The panel that holds the in game Pause Menu Layout for our game

Attributes

- **JButton resume :** The help button. When pressed, the PlayGamePanel is displayed back.
- **JButton settings :** The settings button. the SettingsPanel is displayed.

- **JButton mainMenu :** The main menu button. When pressed, the MainMenuPanel is displayed. Also the last state of the game gets saved upon pressing of this button.
- **JButton exitGame :** The exit game button. When pressed, the game gets closed. Also the last state of the game gets saved upon pressing of this button.
- **JLabel heading:** The label that displays the message “Game Paused”.

Constructors

- **public PauseMenuPanel() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.
- **public void updatePanel():** The method that updates the panel to display the changes on the screen.

3.4.31. SettingsPanel Class

The panel that holds the Settings Screen Layout for our game

Attributes

- **JButton back :** The back button. When pressed, the previous panel is displayed(Main Menu or Pause Menu).
- **JButton soundNo :** One of the sound buttons. When pressed, it toggles sound as closed.
- **JButton soundLow :** One of the sound buttons. When pressed, it toggles sound as low.
- **JButton soundHigh :** One of the sound buttons. When pressed, it toggles sound as high.
- **ArrayList<JButton> themeButtons :** The list of toggle theme buttons. This is kept as a list to provide maintainability, if the game gets updated with new themes.
- **JLabel heading:** The label that displays the heading to this panel (Settings).
- **JLabel sound:** The label that displays the message “Sound”. This is one of the subheadings.
- **JLabel themes:** The label that displays the message “Themes”. This is the other subheading

Constructors

- **public SettingsPanel() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.
- **public void updatePanel():** The method that updates the panel to display the changes on the screen.

3.4.32. CreditsPanel Class

The panel that holds the Credits Screen for our game

Attributes

- **JButton back :** The back button. When pressed, the previous panel is displayed(Main Menu).
- **ArrayList<JLabel> names :** The list that holds labels for the names of the staff. This is kept as a list to provide maintainability, if the developers and staff changes during the development and support period of the game.
- **JLabel heading :** The label that displays the heading to this panel (Credits).
- **JLabel subHeading :** The label that displays the subHeading to this panel (Developers and Staff).

Constructors

- **public CreditsPanel() :** Empty constructor that initializes values to their specified initial values.

Methods

- **public void paintComponent(Graphics g):** Specifies how the panel should be displayed (painted) on the screen.
- **public void updatePanel():** The method that updates the panel to display the changes on the screen.

3.4.33. Main Class

Main class is responsible for connecting the Model and Controller with the View. Additionally it is responsible for setting up the Game Engine and starting a thread for the GameEngine. In this Game Engine thread, the run method of Game Engine is called 60 times every frame.