Bilkent University

Department of Computer Engineering

CS319 Term Project

Rush Hour Game

Design Report (Iteration 2)

Aldo Tali

Barış Can

Endi Merkuri

Hygerta Imeri

Sıla İnci

Instructor: Eray Tüzün

Assigned TA: Muhammed Çavuşoğlu

Design Report

December 12, 2018

This report is submitted (softcopy) to GitHub repository in partial fulfilment of the requirements of the Term Design Project of CS319 course.

Contents

Design Report

1. Introduction

1.1. Purpose of the System

Rush Hour is an entertaining puzzle game in which you try to arrange the other vehicles by sliding them in such a way that you get your car through the exit. There will be several levels (easy, medium, hard) with several dimensions (6X6, 8X8, 10X10) in order to maximize the satisfaction of the game. However, you have to have a certain amount of stars to unlock the harder maps so that there is continuous progression. If the player is stuck and can not get any stars, there will be daily star rewards to keep the interest of the player up for the game. Moreover, with the timer mode and gaining some stars, we aim to make the game more challenging and fun for the users that are already familiar to this type of puzzle games. There will also be several game themes and car skins to increase variety through the game by using some stars so that the game feels more refreshed.

1.2. Design Goals

Usability

Rush Hour is expected to be entertaining for all users. That is why it will have a simple and easy to understand user interface such that you do not need to search for things even in your first trial. For example, the progression data you would like to find, like how many stars you have or which skins you own will always be accessible through the dashboard from the main menu without the need to search for them. There will be a tutorial at the beginning of the game in order to introduce the game and its mechanics. Also, tutorials will always be accessible through the main menu in case the user wants to re-examine some features in the game.

Reliability

The game will not require an internet connection, and the player's data will be kept in the local storage by using the Java Serializable interface. Therefore, it is expected that it will reduce the chance of security problems. The game will be saved automatically after completing each level, but if the player leaves in the middle of the game, the progress will not be saved. Because the game will have an undo option, players' moves will be saved as long as they do not leave the game. The game will not have major crashing issues or bugs that prevent the player from playing the game.

Performance

Since this is a game, it should work smoothly to not spoil the flow of the game. The game will be implemented in a way that the player does not wait for more than 2 seconds to perform the input and see the result. We will use several libraries to make the game as optimized as possible.

Supportability

Rush Hour will run in a Windows operating system and require Java. Classes and methods will be coded in an organized manner so that future problems can be solved quickly.

Extensibility

Rush Hour will be implemented in a way that features like new maps or car skins can be added without changing many high classes in order to keep the game fresh. In addition, the code will be easily understandable for outside coders via good architecture and commenting.

Portability

Since there is no internet connection in Rush Hour, portability is very important in order to open your saved games from another computer. The game will have a small size so that it can be easily transferred into another computer without any problem. Moreover, because the saved files will be kept in a way that they can be directly loaded as objects, transferring or backing up the save will be very easy.

Reusability

We will use the MVC design pattern to make the classes that have different functions corresponding to the Model, View and Controller, as independent as possible so that they can be easily modified later. In addition, we will use some GUI frameworks which are quickly understandable and changeable for future projects.

2. High-level Software Architecture

In this section, we discuss the high-level software architecture of our project. The following subsections give detailed information about the organization of the components of the system. Our high-level architecture represents the MVC model in which 'Model' implements the central data structure and control objects dictate the control flow [1].

2.1 Subsystem Decomposition

Subsystems are classified into three different types: the model subsystem maintains domain knowledge, the view subsystem displays it to the user, and the controller subsystem manages the sequence of interactions with the user [1].

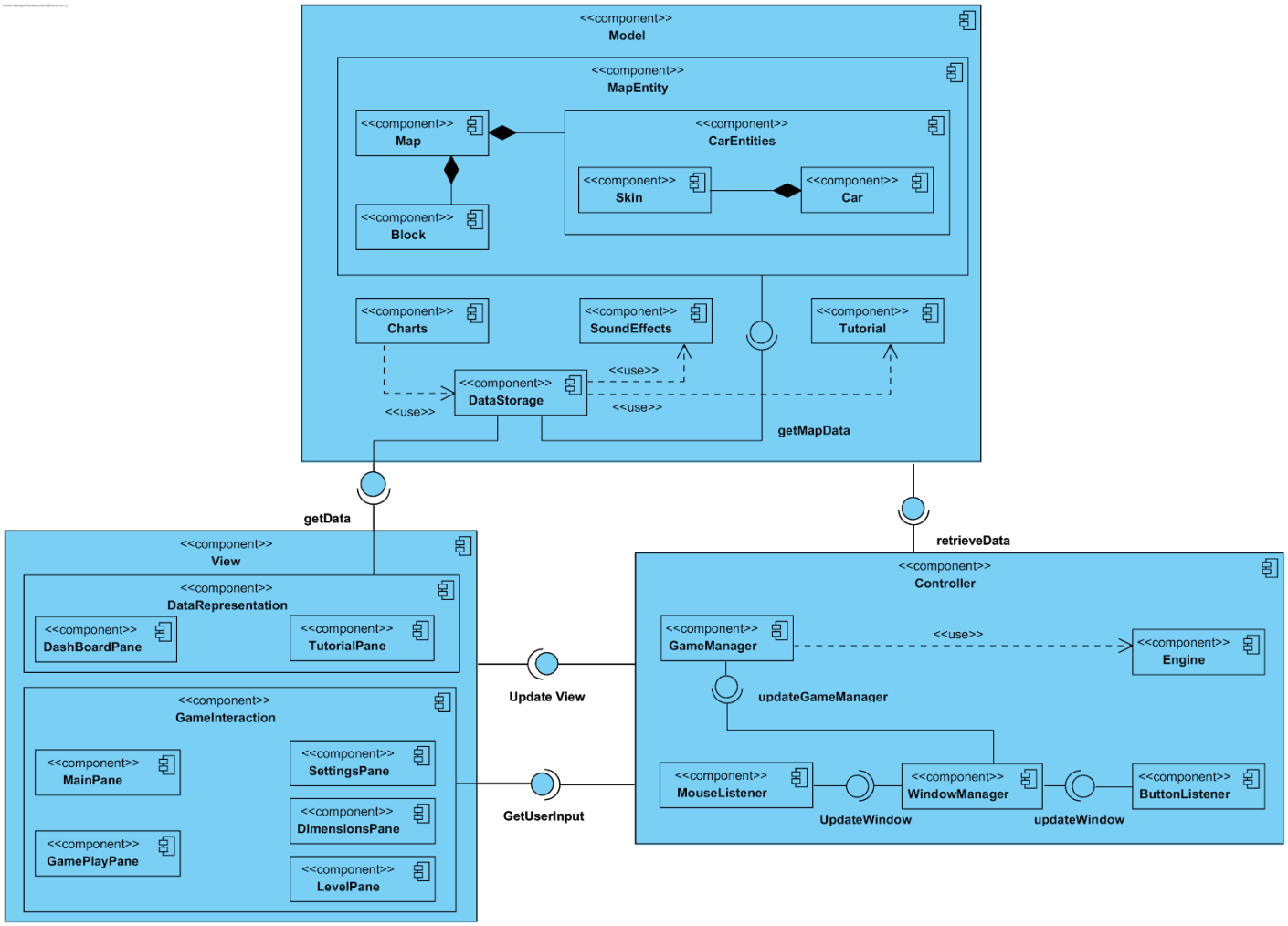

Fig. 1. Rush Hour Component Diagram

Above, the entirety of our system is shown in the notation of the component diagram. The main purpose of this diagram is to introduce what systems are in relation and which design pattern our game uses. The advantage of separating the system into subsystems is that when in need, we can make changes to the game without interrupting any working parts of the system.

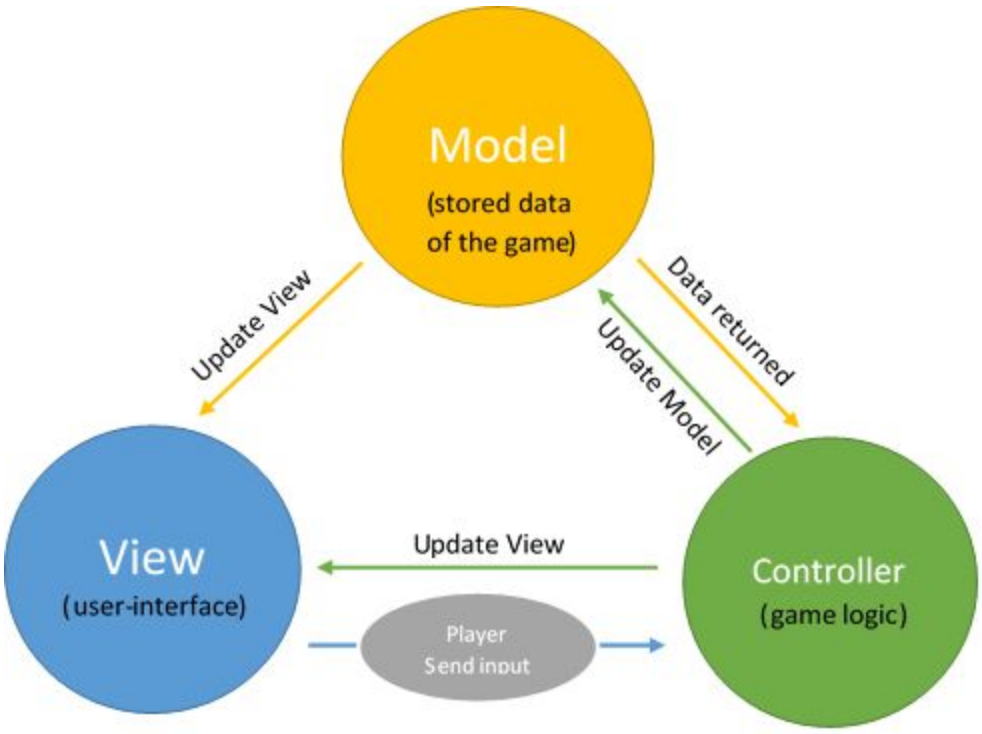

Fig. 2. MVC Logic Representation

For the design pattern of this project, we have chosen to use and apply *Model-View-Controller* (MVC) (as represented in the diagram above). While designing our component diagram we have divided our subsystems in order for it to be fitted with the MVC design pattern.

The primary reason we decided to use this particular design pattern is that it makes it possible for us to make the three subsystems independent from each other. In this way, we will be able to handle the changes in the design of the user interface or the addition of new features more easily by altering the implementation of only one of the subsystems rather than reprogramming the whole game. Another reason we have chosen MVC is its compatibility with the way the objects in our game are defined and interact with each other. By using this design pattern we will also be able to easily develop the game in the Implementation stage and fix any kind of software bug by firstly defining the subsystem that contains the problem.

The system of Rush Hour proposed in this report is divided into three main components which match the MVC model.

- The Control of our design is represented by a higher level component called the Game Manager which controls the Window Manager the Engine of the Game and the File manager. By using the Game Manager we provide for a logical connection of each component in a scalable form.
- The View model is handled by the Window Manager in the Object design. In the proposed system there is one main screen that provides for the user interface with which the end user interacts. Within this main screen, only the inner panels are changing which form the each of the separate game screens for the interface.

- The Model of Rush hour is given by the low-level files that need to be accessed. Here we have included Game entities such as the Maps, The cars and the blocks that compose each map. All these are accessed by the File Manager to provide for the communication with the Game Manager.

2.2 Hardware/Software Mapping

Rush Hour virtual board game will be implemented by making use of Java programming language. We will use JavaFX libraries for implementing the desktop GUI needed for all the screens of the game. As such the software requirement for playing “Rush Hour” will be the Java Runtime Environment in order for the execution of the game to be initiated and kept throughout. Also, given that the above-mentioned libraries will be used, “Rush Hour” will strictly require the Java Development Kit 8 or a newer version (8 or above) since the JavaFX library was introduced and made available in these versions of Java.

Regarding the hardware requirements “Rush Hour” makes use of mouse events both throughout the gameplay and the GUI control. This means that the main and only I/O device that will be needed in the game is the mouse. The user will make the screen selections, arrange his preferred settings, themes and personalize his own gameplay by clicking on the designated positions on the game. Similarly, the user will use the mouse only, to drag the cars up, down, forward and backwards. Given all of the above then our game’s system requirements will be minimal since they will need only a running computer that has the adequate software to compile the game and a mouse to interact with it while it is running.

The last requirement deals with the storage system that is related to the DataStorage. In order to store the objects that are required, we will use the Serializable interface and the storage of all intermediate media files will be either in JPeg or Png formats. As such the storage system will not need any internet connection nor the need to communicate to a database storage.

2.3 Persistent Data Management

Rush Hour will store its data content, in the end, user’s hard drive and not in a complex database storage system. This design decision is particularly influenced by the static and minimal nature of the proposed virtual version of the board game. Rush Hour will need to store the maps of the gameplay which internally will be text files containing the corresponding matrix representation of the respective game map. These files by nature are static and will not be changed. As such the game will have a fixed number of maps. Along with the maps, the game system will store the sound effects for the win, lose and/or time-out scenarios.

Again these are minimal and will not be changed so the storage of these files will again be in the end user's hard drive in the format .waw or .mp3. The image files, on the other hand, will be stored in JPeg and PNG formats. The last type of information that the game system stores is the user preference information for the game. Given that the game has only one user throughout and there is a fixed number of possible preferences one can choose in Rush Hour game even this information will be logged out in JSON format so that it is structured and easy to access depending on the type of preference needed at runtime. Altogether these design decisions, provide for a persistent data management as they keep the information organized, minimal and marshalled into a easy to access fashion.

The game system proposed by Royal Flush takes into consideration that there is a limited number of files that need to be stored on the system. The files needed for Sound Effects (only 3, one for the game win, one for the game loss and the last for in-game effects), for Settings (only 5 themes), for the Maps (50 maps) and the Cars (13 different cars) is a finite number which is relatively small both when projecting the file sizes as well as when projecting the number of operations needed to retrieve, manipulate and integrate the files into our game. Since all these files are pertaining only to their respective modules, it is easy to directly access them on the file system. If we were to use a database, however, we would be adding overhead to a job that as simple as to be solved by accessing the file system. This discussion does not involve runtimes as the database would be small and fast regardless if we were to keep an index or not to the files. Also considering that the game was designed to fit into a "closed box" type of model, meaning that the user himself cannot in any way add more of these files into the game, the use of a database is clearly an overkill for this design. All this taken into account the group preferred to leave the usage of a database out and deal only with the FileSystem.

2.4 Access Control and Security

By design, Rush Hour will be a game that will have only one user after its compilation. Ideally, anyone will be able to play the game and since there will be no login system or user specific sensitive data storage in the game. In other words in Rush Hour "a user" is technically one computer or a game installation in a particular device and no information is kept related to third parties or individuals. As such there will be no actual precautions or actions taken to prevent data leakage due to the absence of user data. Similarly as discussed in the previous sections the storage is not done in a complex database system nor in a cloud provider. This eliminates need to protect the application from potential internet/network related malware and from the threat of having to take into consideration miscellaneous or malicious acts that violate security. Given the simplistic nature of the game, none of the stored files mentioned in section 2.3 represents a

storable that could lead to a critical security issue which in turn means that the current design of the game requires no access control.

Access Matrix

The access control matrix for our game is given below. We have not included the classes in which the player has no control over. The classes and operations that are included represent the methods that are evoked when the player performs some actions in the game. For example, GameManager is created and its initialize() method is called when the player opens the game when the player moves a car in the grid PlayGamePane's updateCar() method is evoked.

	Car	Engine	GameTimer	GameSolver	WindowManager	PlayGamePane	DashboardData	GameManager
PLAYER	changeCarSkin() move()	undo() getHint() moveCar() setVolume() resume() reset()	setTimerOn() startClock()	generateHint()	setCurrentColor() updateMiddlePanel()	updateCar()	setUserPicture()	<<create>> initialize()

Fig.3. Access Matrix

2.5 Boundary Conditions

Rush Hour game will be deployed to the end user by having a .jar executable. The decision to have .jar instead of .exe execution files comes due to the fact that operating systems like Linux do not run the same executables as windows. Nevertheless by making use of the .jar executable the running of the game is based on the Java's runtime environment which internally provides with the necessary abstraction form the operating system. This way the game is operable in both cases and it makes its deployment much easier for "Royal Flush" since Java is the chosen implementation language. Along with supportability, this adds to the portability as well. The executable is easily portable in different devices and as such can be run accordingly.

The game is terminated once the "Exit" button is pressed. Upon close, the game will keep the progress of levels and dimensions in the game. This will not be the case however if any of the storage files is corrupted or cannot be accessed to improper user changes in the storage folder. The game resets however if the .jar executable is given again from the beginning to the device. Rush Hour might incur other possible errors in the game, being the sound effects or the images are corrupted. In these cases, the game will function without the sound effects and will use drawables to substitute for images. The GUI, however, is expected to be fundamentally more basic as compared to the image based content display. The last corruptible pieces of files include the maps themselves which means that the respective map won't be able to be displayed.

In the case where every map gets corrupted and no actual proper access can be done the game collapses. Upon collapse, the user game preference JSON data is deleted and the game should reset. It will, however, require either the fixing of the corrupted files by the user (example: substitute them with the uncorrupted versions) or the .jar executable needs to be replaced for the game to start from the beginning and all previous information is lost.

3. *Subsystem Services*

In this section, we discuss the subsystem services of our project. Because the goal of decomposition of the system is for the people to understand and work on the project better, we are able to get into the detail of the subsystems individually.

3.1 UI Subsystem

The following the is the visualization of the UI Subsystem.

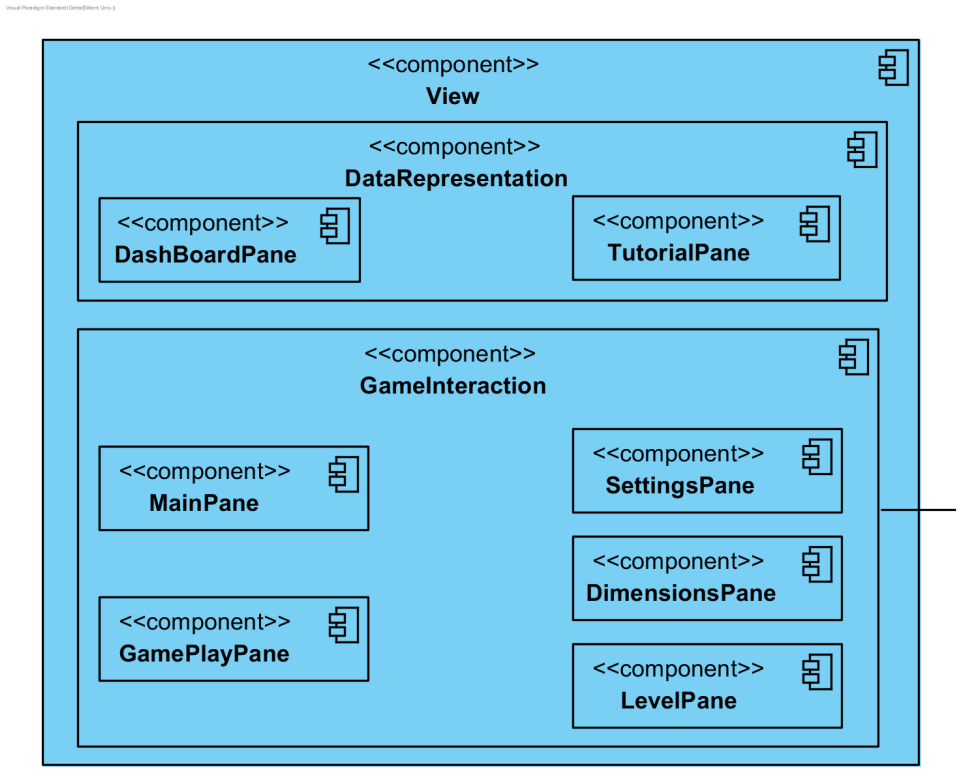

Fig. 4. View Component of Rush Hour

The **UI Subsystem** is responsible for all the screens shown throughout the game. The screens are connected to each other not by require-provide and/or dependency relation, but via controllers that manages these screens. This subsystem has 7 components which are listed below:

1. MainPane
2. SettingsPane
3. TutorialPane
4. DimensionsPane
5. LevelPane
6. GamePlayPane
7. DashboardPane

The *MainPane* view is responsible for the opening screen of the game. The subflows that connect the other screens are shown on the MainPanel.

The *SettingsPane* view is responsible for displaying the settings view which is the screen that enables user to adjust volume, select timer mode and change the theme of the game.

The *TutorialPane* view is responsible for displaying the tutorial which teaches the user who has never played the game before how the game operates and what are the controls to play the game. This screen also has smaller buttons for going straight to settings and again, adjusting the volume.

The *DimensionsPane* view is responsible for displaying dimensions for the maps the user can select. This screen also has smaller buttons for going straight to settings and again, adjusting the volume. The screen also requires the amount of stars from the DataStorage in order to show them below the each dimension.

The *LevelPane* view will be used to display a list of the levels that the user is allowed to play, i.e. the levels that the user has unlocked so far in the dimension that the user chooses. This panel will also display the corresponding number of stars gained in each of the levels that the user has completed.

The *GamePlayPane* view is responsible for displaying the map which is requested from the GameManager. The screen has smaller buttons for asking hints, for undoing moves that has been made previously and reset which are requested from the GameManager.

The *DashboardPane* view will be used to show a more informative representation of all the information related to the users and his or her statistics related to the game. It will show a number of charts, graphs, the total number of stars the user has collected and the user picture. In this view, the user will also be able to change the skin of his car.

3.2 Controller Subsystem

The following the is the visualization of the Controller Subsystem.

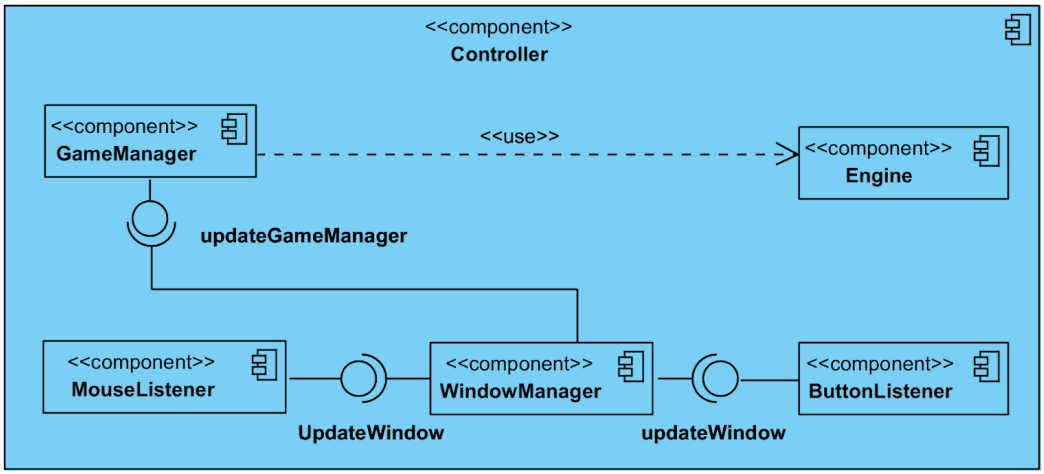

Fig. 5. Controller Component of Rush Hour

The **Controller Subsystem** is responsible for managing the flow of the decision making of the game. This subsystem has 6 subcomponents which are listed below:

1. MouseListener
2. WindowManager
3. ButtonListener
4. DataStorage
5. GameManager
6. Engine

The *MouseListener* provides the information about where the mouse was clicked on the screen to WindowManager. MouseListener gives the necessary implementation of the MouseMotionAdapter.

The *WindowManager* is the responsible controller for the entirety of the screens which composes our UI Subsystem. The WindowManager requires the information about the location of mouse clicks from the MouseListener and also requires the information about which button is pressed from the ButtonListener. This subsystem provides to the GameManager the access of the flow and control of the screens.

The *ButtonListener* gives the proper implementation of the ActionListener interface and provides the information of whether a button is pressed and if so which button is pressed to the WindowManager.

The *DataStorage* is responsible for all the storage and managing the operations related to showing the data on the Dashboard screen. It also has a dependency relation with the GameManager regarding that the GameManager controls all the data flow that requires for the game to operate accordingly.

The *GameManager* is responsible for the operations the game needs to complete in order for it to work. It provides the necessary information about the map and the user request regarding the gameplay to the Engine. The GameManager requires the information about the maps, charts, sound effects etc. from the DataStorage. It also requires needed details regarding the mouseClick and buttonPressed from the WindowManager.

The *Engine* is responsible for all the in-game mechanics like moving the car, setting the timer or calculating the stars for a map. While the original maps are kept in the DataStorage, Engine will select a map from DataStorage with the help of GameManager according to the player's input, make moves for the player using the Car objects inside the selectedMap, save the moves of the player, call Undo method and decide if the game is lost or won. Engine can also set the volume and play the SoundEffect.

3.3 Model Subsystem

The following the is the visualization of the Model Subsystem.

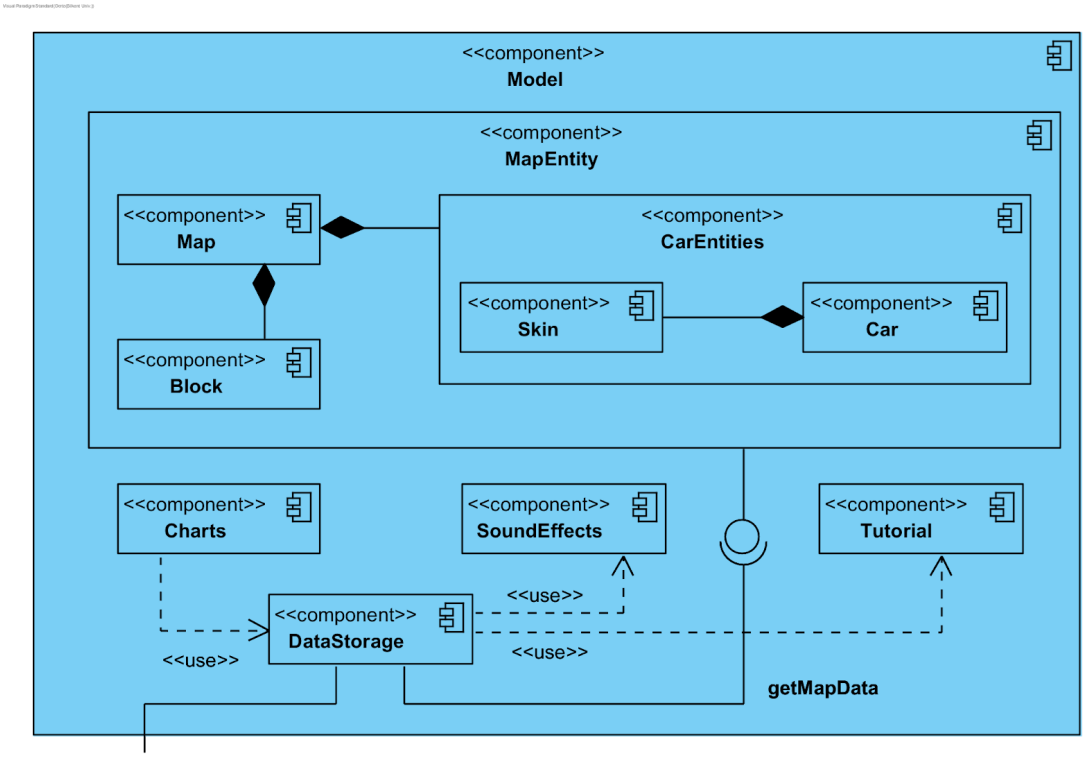

Fig. 6. Model Component of Rush Hour

The **Model Subsystem** contains the classes that are used to store and manipulate some state. This subsystem updates DataStorage. This subsystem has one main component which is the *Storage* component. This subcomponent is composed of *Charts*, *SoundEffect*, *Tutorial* and *MapEntity*. *MapEntity* has 3 components; Map, Block and Car Entities. Car Entities is composed of two components; Car and Skin. In total the *Model* subsystem has 6 subcomponents which are listed below:

1. Storage
    1.1. MapEntity
        1.1.1. Map
        1.1.2. Block
        1.1.3. CarEntities
            1.1.3.1. Skin
            1.1.3.2. Car
        1.1.4. Charts
        1.1.5. SoundEffects
        1.1.6. Tutorial

The *Storage* component serves as a container and it contains all the model objects.

The *MapsEntity* is a container that includes all model objects of the map.

The *Maps* component represents the map, the blocks and cars that make up the configuration of the map, as well as level, total number of stars collected for each map and the corresponding operations that can be carried out.

The *Block* component represents the parts of the map composition. A block is an occupied or free part of the map that has coordinates and includes its operations.

According to the organization of our model objects, it can be seen that the map depends on blocks and CarEntities.

The *CarEntities* subcomponent is composed by *Car* which uses a *Skin.*

The C*harts, SoundEffects, Tutorial* components provide an interface to *DataStorage* and include their operations as well.

3.4. 3-Layer Architecture

The following diagram is the visualization of our system regarding the 3-layer architecture style which includes layers; presentation, application and data.

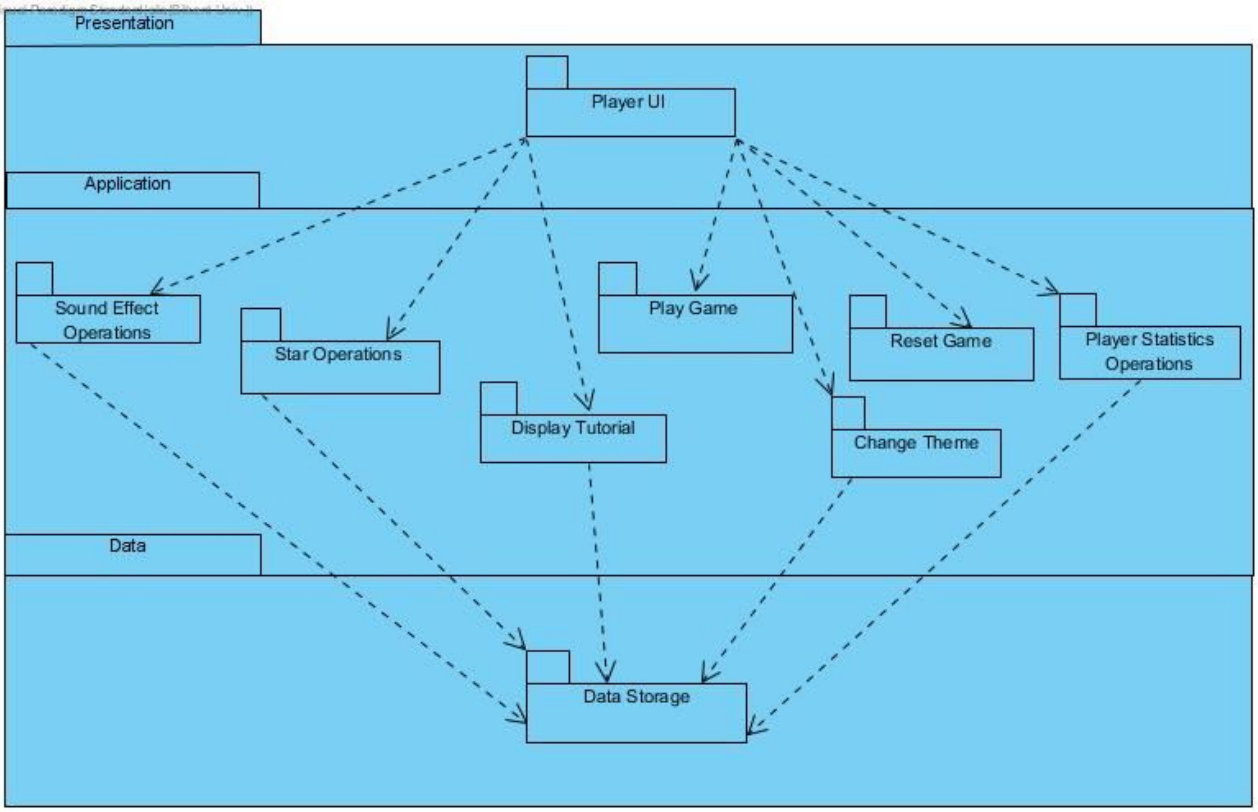

Fig.7. 3-Layer (Opaque) Architecture Representation

The main idea of the 3-Layer Architecture is that it does not allow the client, the user or else access the server, database or the file system, in this case, directly. The MVC architecture allows the connections between its 3 components while the structure of the 3-layer architecture is linear which indicates the hierarchy between the layers. Our 3-layer architecture is of an opaque type, meaning that each layer can only call operations from the layer below. This is done to support our design goals: extensibility and reusability.

Since there are no other actors than the player, we only have a user interface for the player of the game. By this interface, the player can access several operations such as sound effect operations, star operations, player statistic operations etc. These operations make muting the sound effect, adjusting the volume, receiving and using stars, changing themes, displaying the charts which are related to the performance of the player regarding the gameplay, available to the user. The packages which depend on the Data Storage are *Sound Effect Operations* because the sound files are kept there, *Star Operations* because the track of the stars are kept in the storage, *Display Tutorial* because the file which what is shown to the user as a tutorial is stored in this component, *Change Theme* because the different themes are also kept in the storage, *Player Statistics Operations* because all of the achievements of the player, high scores, the cars skins they have won is stored in the storage as data.

4. Low-level Design

4.1. Object Design Trade-offs

Maintainability vs Space:

In order to make the game more maintainable for future revisions, we decided to save most of the constants used, as properties of the objects. In this way, the size of each object will increase which can make further iterations less efficient if other features are added.

Performance vs Maintainability:

Since we have decided to use Java to implement the game, its performance will not be as good as games implemented in lower level languages such as C or C++. The reason we chose Java is that as all the members are proficient in Java, so it will be easier to maintain the software and modify it. This programming language will be also helpful to increase the reliability of the game, decreasing the number and frequency of crashes.

Rapid Development vs. Functionality

This game is expected to have a development period of approximately two months. For the time given, not more than eight features are expected to be implemented. Therefore, this game won't be able to provide the player with the full functionalities that would be implemented if the development was given more time. In order to come up with a fully working game (with crucial features), we have to leave some of the extra features unimplemented.

Functionality vs. Usability

This game is expected to have an optimal number of features, enough that the game is enjoyable and easy to be understood and managed by players aged 8 and above. Therefore, we traded off functionality for usability. Adding more features to the game would make it more complicated and therefore less usable by children.

Efficiency vs. Portability

This game is written in Java programming language. For this reason, it can be run on any hardware that has a compliant JVM (Java Virtual Machine). This game can be used in operating systems other than the one in which it was created without requiring major rework (other than installing JVM, if not provided). The fact that we use Java to implement our game makes it (the game) more portable but at the same time makes it less efficient compared to a game that is implemented using other programming languages such as C++. So, in this

game, we traded off efficiency for portability, because we want more players (using machines with different OS) to play our game.

4.2. Final Object Design

The following diagram is the complete object design diagram with all the classes, their attributes and operations expanded. The classes are divided in layers which are explained in detail later in the report.

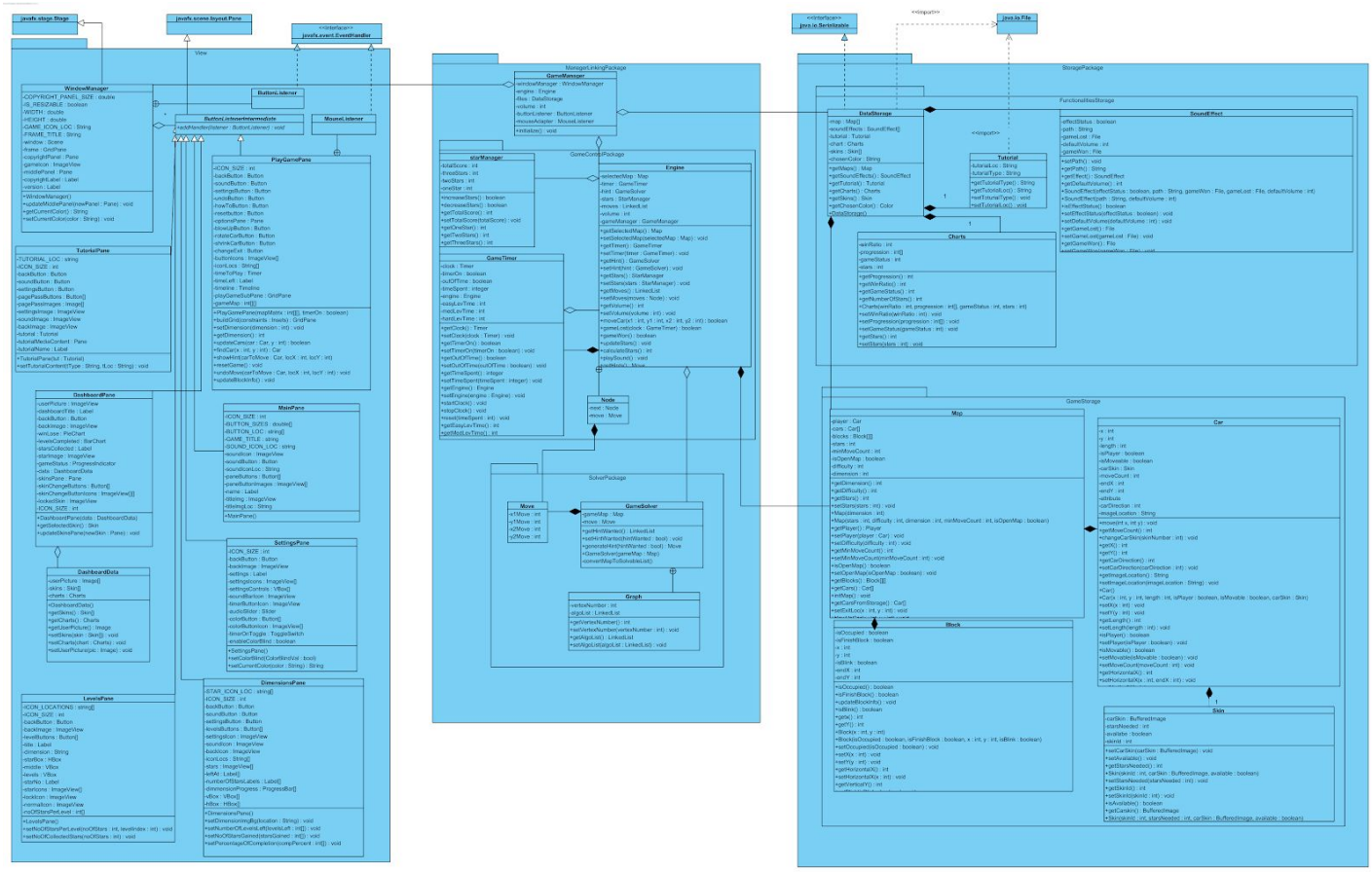

Fig. 8. Full Class Diagram of the System

Final Object Design Abstraction:

Furthermore, the following diagram is an abstraction of this class diagram containing only the classes and their relations is also provided.

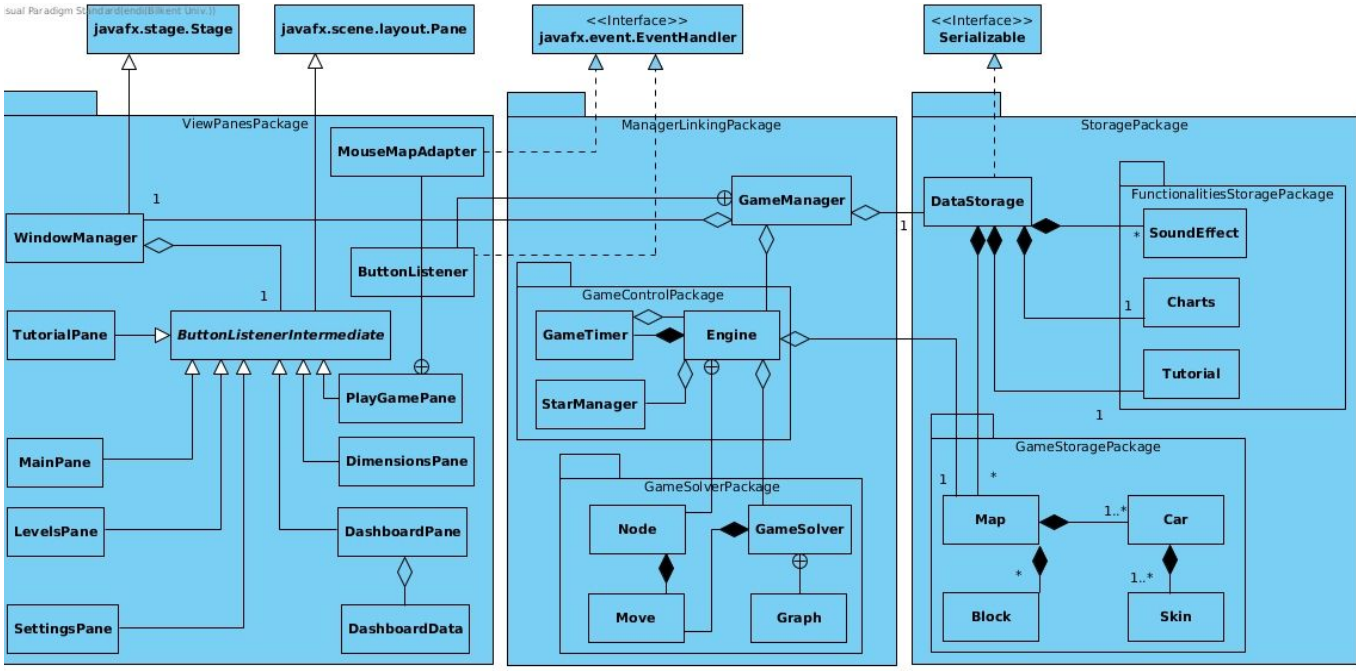

Fig. 9. Abstract Class Diagram

4.3. Packages

4.3.1. Internal Packages

The system has been divided into several packages to help with the division of the system as a whole into groupings that facilitate the implementation of the game. We have divided the game into seven packages where three of them behave as higher level packages that encapsulate the larger levels of the system. The other four are sub packages that encapsulate the subsystems of the higher level packages. Below we have provided a detailed description of these packages:

ViewPanesPackage

This package encapsulates all the screens that need to be displayed throughout the interaction of the player and the game. The package is used to provide for all the GUI that the game will have to display.

ManagerLinkingPackage

This package serves as a higher level package that provides for the necessary connections between the game controller the views and the objects of the game. Its main purpose is to serve as a connector between the other packages of the system.

GameControlPackage

This is a sub package of the ManagerLinking and it serves to control the game flow and to make the necessary requests for any possible change in the game as the interaction with the player progresses.

SolverPackage

This is a sub package of the ManagerLinking and it serves to provide for the necessary functionalities that are needed to apply the algorithms of the game solving and to get the list of possible moves that finish the level.

StoragePackage

The storage package provides for an encapsulation of all the game objects and all the external files that are required in the game.

FunctionalititesStoragePackage

This is a sub package of the StoragePackage and it provides the necessary storage structure for all the external files need for functionalities of the game such as the sound effects, the tutorial or the provision of user charts.

GameStoragePackage

This is a sub package of the StoragePackage and it provides the necessary storage structure for all the external files need for the game objects such as the map, the car, the block or the skin.

4.3.2. External Packages (Java Packages)

In order to provide for the implementation of the game, it will be needed to make use of four main external java packages. The external packages encapsulate the utilities needed for the implementation of the UI part and the main game mechanism. Below we have provided a detailed description of these packages:

javafx.scene

This package will be used to implement the classes that will create the static GUI elements like panes and frames. The main classes that we will use from this package are the GridPane and Pane class.

javafx.event

We will use this java package to make the classes created by the javafx.scene package functional by adding event listeners which is the main purpose of this class. The interfaces related to event listeners present in this package will be useful when implementing the graphical interface.

java.io

This Java package is specialized in handling file input-output operations. We will use it to store the data that need to be loaded after the user exits the game. The main part of this package that will provide a simple way of storing the data is the Serializable interface, which will be used to store data such that it can be loaded and used directly as objects.

java.util

The java.util package contains a Timer class that will be used to implement the GameTimer class of our game. The GameTimer class will then be used in the Timer mode of the game.

4.4. Class Interfaces

The above class diagram is divided into three parts. These are just parts of the full class diagram to create a clearer view of the whole diagram rather than a strict division into subsystems. The corresponding layer names are just a simple description of all the classes included.

4.4.1. User Interface Layer

In this section, we discuss the user interface classes of our project. In this layer, there are seven panes, one frame(scene) and one data collection for the dashboard.

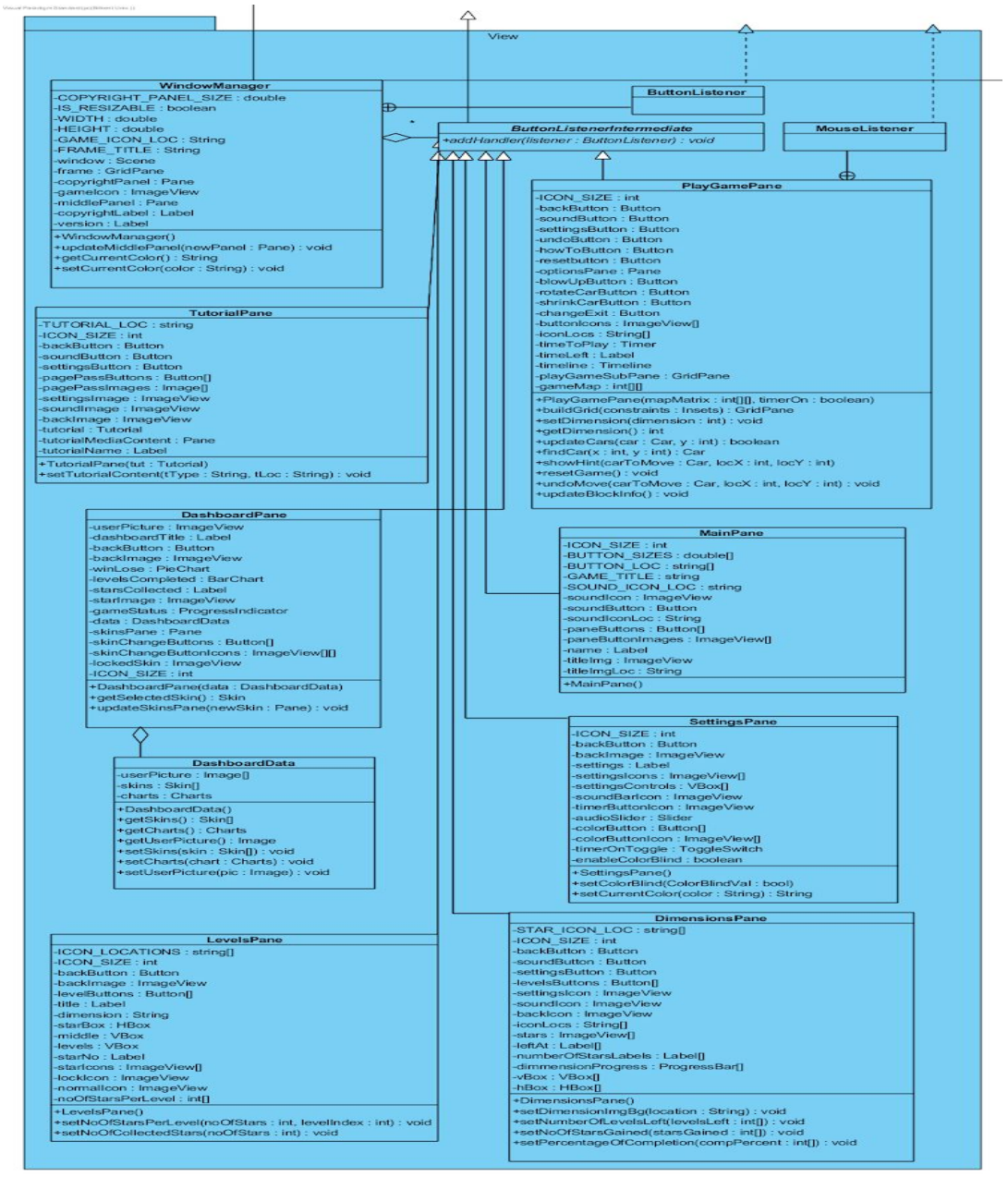

Fig. 10. User Interface Layer Classes

WindowManager:

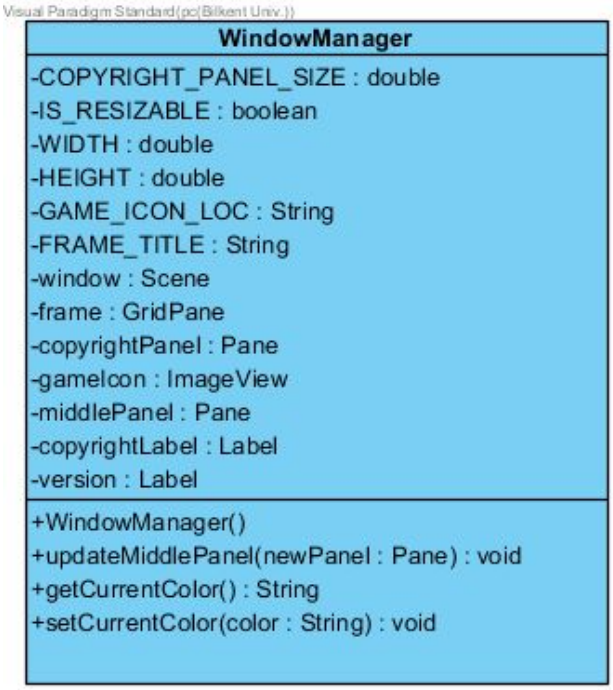

Fig. 11. WindowManager Class

Attributes:

- **private final double COPYRIGHT_PANEL_SIZE:** Size of the copyright pane.
- **private final boolean IS_RESIZABLE:** isResizable will keep the resizability information of the frame.
- **private final double WIDTH:** Stores the width of the game frame.
- **private final double HEIGHT:** Stores the height of the game frame.
- **private final String FRAME_TITLE:** Frame title will keep the title of the frame in the game.
- **private final String GAME_ICON_LOG:** Keeps the icon found in the top leftmost corner of the frame.
- **private Scene window:** Scene for our JavaFx game.
- **private GridPane frame:** Our frame consists of two panes, a middle pane which changes when a new screen opens, and a copyright pane which consists of two labels.
- **private Pane copyrightPanel:** copyrightPanel will keep the bottom-most Panel of the screen.
- **private ImageView gameIcon:** Icon of the game.
- **private Pane middlePanel:** middlePanel will keep the middle Panel of the screen. This is the panel that keeps changing based on user operations.
- **private Label copyRightLabel:** copyrightLabel "Developed by Royal Flush"
- **private Label version:** Version of our game.

Operations:

- **public WindowManager():** Constructor of the WindowManager.
- **public void updateMiddlePanel(Pane newPanel):** Updates the central GUI panel to the application.
- **public String getCurrentColor():** Returns the current selected theme of the user.
- **public void setCurrentColor(String color):** Sets the current selected theme of the user.

TutorialPane:

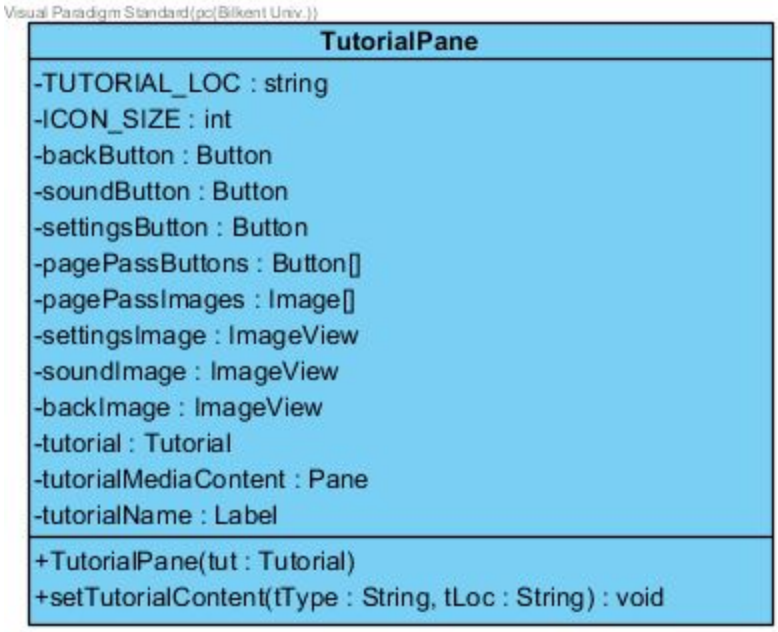

Fig. 12. TutorialPane Class

Attributes:

- **Private final String TUTORIAL_LOC:** The location of the tutorial.
- **private final int ICON_SIZE:** Size of the icons.
- **private Button backButton:** The Tutorial Panel will store the back button which will go to main screen.
- **private Button soundButton:** Players can mute the game anytime with soundButton.
- **private Button settingsButton:** settingsButton will go to settings screen.
- **private Button[] pagePassButtons:** There will be several tutorial images and pagePassButtons will change those images.
- **Private Image[] pagePassImages:** Images of pagePassButtons.
- **private ImageView settingsImage:** This will store the icon image that displays as a symbol for settings.
- **private ImageView soundImage:** This will store the icon image that displays as a symbol for sound.

- **private ImageView backImage:** This will store the icon image that displays as a symbol for back button.
- **private Pane tutorialMediaContent:** This will store the tutorial to be displayed on the screen for the gameplay. There will be several media content and two buttons in it.
- **private Tutorial tutorial:** Our tutorial's contents.
- **private Label tutorialName:** Tutorial title of the screen.

Operations:

- **public TutorialPane():** Constructor of the TutorialPane.
- **public void setTutorialContent(String tType, String tLoc):** Setting tutorial contents' type and location.

MainPane:

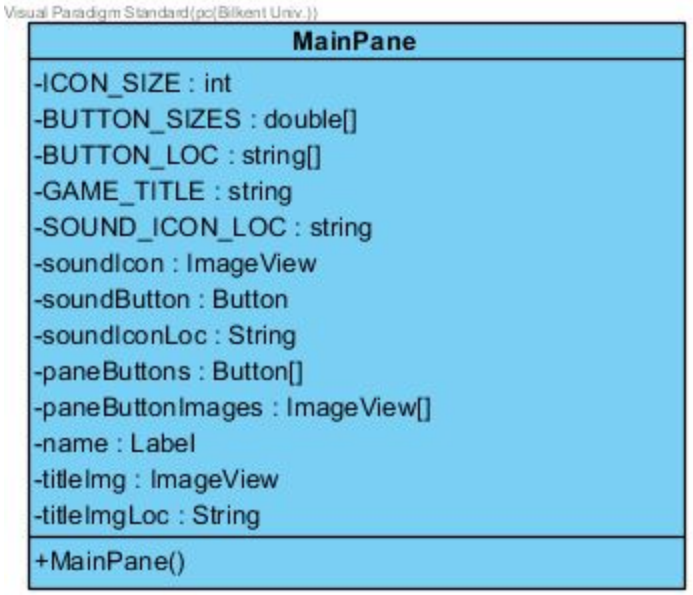

Fig. 13. MainPane Class

Attributes:

- **private final int ICON_SIZE:** Size of the icons.
- **private final double[] BUTTON_SIZES:** Size of the main screen buttons.
- **private final double[] BUTTON_LOC:** Location of the main screen buttons.
- **private final String GAME_TITLE:** Name of our game.
- **private final String SOUND_ICON_LOC:** This keeps the location of the sound icon in the play screen.
- **private ImageView soundIcon:** The image holds the icon that is used for showing the sound.
- **private Button soundButton:** Sound button for muting the game.

- **private String soundIconLoc:** Location of the sound icon.
- **Private Button[] paneButtons:** There will be four buttons in the main pane. playButton which leads to Dimensions screen, tutorialButton which leads to Tutorials screen. dashboardButton which leads to Dashboard screen and settingsButton which leads to Settings screen.
- **private ImageView[] paneButtonImages:** Images of the main screen buttons.
- **private Label name:** Game title label.
- **private ImageView titleImg:** Image of the title.
- **private String titleImgLoc:** Location of the titleImg

Operations:

- **public MainPane():** Constructor of the MainPane.

LevelsPane:

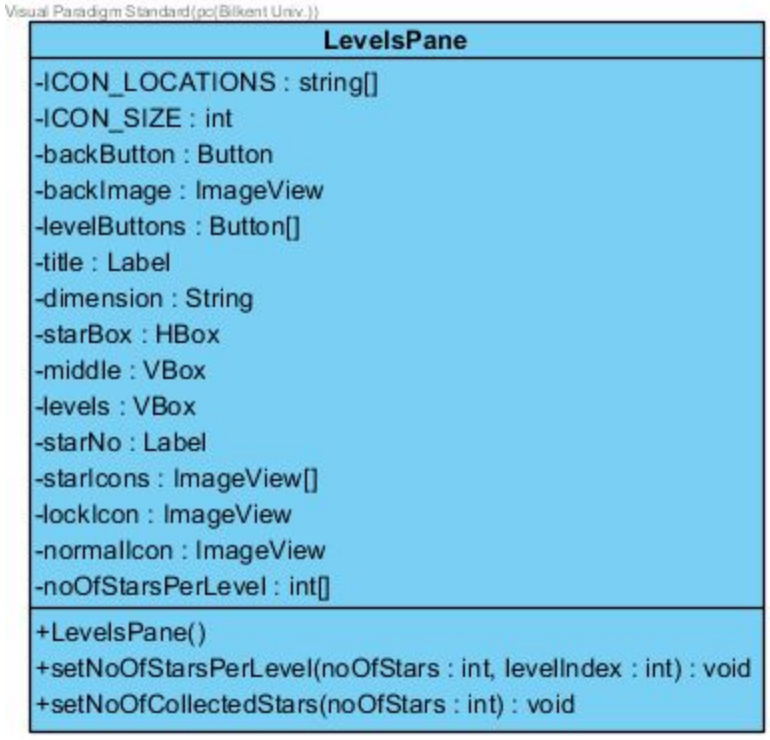

Fig. 14. LevelsPane Class

Attributes:

- **Private String[] ICON_LOCATİONS:** Locations of the button icons.
- **private final int ICON_SIZE:** Size of the icons.

- **private Button backButton:** The Levels Screen will store the back button which leads to Dimensions screen.
- **private ImageView backImage:** Image of the back button.
- **private Button[] levelsButtons:** Maps' buttons which will lead to Play screen.
- **private Label title:** Name of the dimension.
- **private String dimension:** String for title label.
- **private HBox starBox:** This is just a box that makes the alignment of the stars in the level buttons easier.
- **private VBox middle:** A structure used to keep all the subcomponents of the Pane aligned vertically.
- **private VBox levels:** This VBox is used to keep all the level buttons aligned.
- **private Label starNo:** Number of stars the player has for that particular level.
- **private ImageView[] starIcons:** There will be three star icons which represents how much star does the player has, fullStarIcon, halfStarIcon and emptyStarIcon.
- **private Image lockIcon:** This is the image of locked key that is used on the levels display for blocked levels.
- **private Image normalIcon:** This is the image of that is used by default to display unblocked levels in the game.
- **private int[] noOfStarsPerLevel:** Number of stars per level.

Operations:

- **public LevelsPane():** Constructor of the LevelsPane.
- **public void setNoOfStarsPerLevel(int indexLevel, int noStars):** This method updates the number if stars for the level at index indexLevel.
- **public void setNoOfCollectedStars(int noOfStars):** The Levels Panel will have a functionality to allow the Window Manager set the number of collected stars for a particular level so that this information can be used of the display of the level.

SettingsPane:

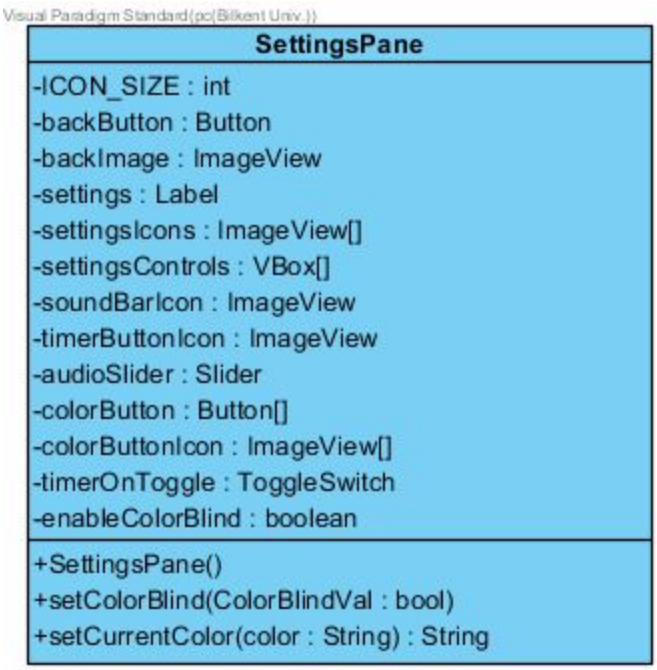

Fig. 15. SettingsPane Class

Attributes:

- **private final int ICON_SIZE:** Size of the icons.
- **private Button backButton:** The Tutorial Screen will store the back button that it displays on the top leftmost corner.
- **private ImageView backImage:** Image of the back button.
- **private Label settings:** Title of the settings.
- **private ImageView[] settingsIcons:** audioImage, timer, themes icons in VBoxes.
- **private VBox[] settingsControls:** Left, center, right area of the settings screen.
- **private ImageView soundBarIcon:** Icon for Slider.
- **private ImageView timerButtonIcon:** Icon for timerOnToggle switch.
- **private Slider audioSlider:** Slider to increase or decrease the sound.
- **private Button[] colorButton:** Buttons for color options.
- **private ImageView[] colorButtonIcon:** Icons for color buttons.
- **private ToggleSwitch timerOnToggle:** This stores the toggle button that turns timer on or of.

- **private boolean enableColorBlind:** Opens colorblind mode.

Operations:

- **public String setCurrentColor(String color):** This method draws the color theme selection in the settings screen. Returns a CSS color to WindowManager to change frame color.
- **public SettingsPane():** Constructor of the SettingsPane.
- Public String setColorBlind(String color): Sets color blind mode.

DashboardData:

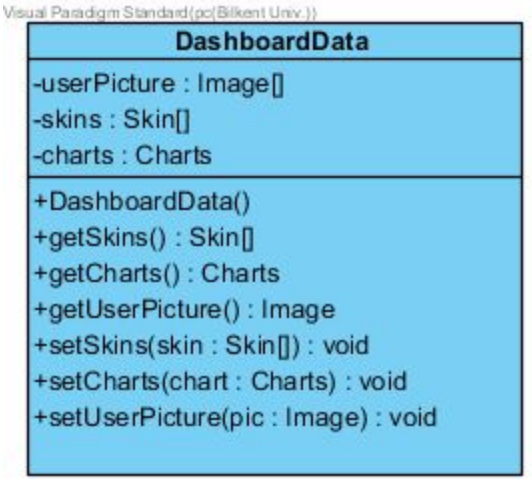

Fig. 16. DashboardData Class

Attributes:

- **private Image[] userPicture:** This image holds the picture that displays on the Dashboard screen on opening it.
- **private Skin[] skins:** This stores the array of skins that are displayed for the user to choose on the dashboard screen.
- **private Charts charts:** This stores the charts' data to be drawn on the dashboard screen.

Operations:

- **public DashboardData():** Constructor of the DashboardData.
- **public Skin[] getSkins():** Returns the available skins to be displayed to the user.

- **public Charts getCharts():** Returns the charts to be displayed to the user.
- **public Image getUserPicture():** Returns the current user picture.
- **public void setSkins(Skin[]):** Sets the available skins to be displayed to the user.
- **public void setCharts(Charts charts):** Sets the charts to be displayed to the user.
- **public void setUserPicture(Image pic):** Sets the current user picture.

DashboardPane:

Fig. 17.DashboardPane Class

Attributes:

- **private ImageView userPicture:** This image holds the picture that displays on the Dashboard screen on opening it.
- **private Label dashboardTitle:** Title of the dashboard.
- **private Button backButton:** backButton will leads to main screen.
- **private ImageView backImage:** Image of the back button.
- **Private PieChart winLose:** PieChart that will show win/lose percentage
- **Private BarChart levelsCompleted:** BarChart that will show completed levels for each dimension.
- **Private Label starsCollected:** Label that will show total number of stars.

- **Private ImageView starImage:** Image for starsCollected label.
- **Private ProgressIndicator gameStatus:** ProgressIndicator that will show completion of the game.
- **private DashboardData data:** This holds all the possible displayable data in the Dashboard.
- **private Pane skinsPane:** This holds the panel of selectable skins.
- **private Button[] skinChangeButtons:** There will be two buttons to change the skins in skins pane.
- **private ImageView[] skinChangeButtonsIcons:** Icons for skinChangeIcons.
- **private ImageView lockedSkin:** Some skins will not be available at the beginning of the game, lockedSkin image will be shown instead of them.
- **private final int ICON_SIZE:** Size of the icons.

Operations:

- **public DashboardPane(DashboardData data):** Constructor of the dashboard.
- **public Skin getSelectedSkin():** Returns the current selected skin to be displayed to the user.
- **public void updateSkinsPane(Pane newSkin):** Updates the skin GUI panel to the application.

PlayGamePane:

Fig. 18. PlayGamePane Class

Attributes:

- **private Button backButton:** The Tutorial Panel will store the back button which will go to Levels screen.
- **private Button soundButton:** Players can mute the game anytime with soundButton.
- **private Button settingsButton:** settingsButton will go to settings screen.
- **private Button undoButton:** undoButton will go to the previous move that the player did.
- **private Button howToButton:** howToButton will give a hint indicating the next possible move for the player.
- **private Button resetButton:** resetButton will back the map to initial configurations.
- **private ImageView[] buttonIcons:** Icons for backButton, soundButton, settingsButton, undoButton, howToButton and resetButton.
- **private Timer timeToPlay:** Keeps the timer that is restricted on the game play. The timer is in a countdown fashion.
- **Private Label timeLeft:** Label that will show the time left for timer mode.
- **Private Timeline timeline:** Timeline for timer mode.
- **private final String[] iconLocs:** Locations for button icons.
- **private final int ICON_SIZE:** Size of the icons.
- **private GridPane playGameSubPane:** This keeps the playGame panel with the map grid and the moving car
- **private Pane optionsPane:** This pane includes the helper functions that can be done in the game.
- **private Button blowUpButton:** a “blow-up car” button that disappears one car from the map.
- **private Button rotateCarButton:** a “rotate car” button that rotates one car from its center.
- **private Button shrinkButton:** a “shrink” button that makes a car one block-size smaller.
- **private Button changeExitButton:** a “change exit” button to change the exit of the game by one block size.
- **private int[][] gameMap:** Map of the game as a two-dimensional array representing blocks.

Operations:

- **public PlayGamePane(Map map, timerOn boolean):** Constructor of the GamePlayPanel.
- **public GridPane buildGrid(Insets constraints):** Creates the playable grid in the GamePlayPanel.
- **public void setDimension(int dimension):** Sets the dimension to 6, 8 or 10.
- **private int getDimension():** Returns the dimension number.

- **private void updateBlockinfo():** When a car changes position, this method will update the block info that the car moved.
- **public boolean updateCarPos(Car car, int x, int y):** Changes the position of a particular car by setting it to x and y.
- **public Car findCar(int x, int y):** Finds and returns the car that is in the block with coordinates x and y. If there is no car there it returns null.
- **public int[] showHint(Car carToMove, int locX, int locY):** After the player presses hint button, the gameSolver class will be initialized and showHint method will be called. The return type will be integer array representing x and y.
- **public void resetGame():** Returns the selected map to its initial configurations.
- **public void undoMove(Car carToMove, int locX, int locY):** Undos one move of the player.

DimensionsPane:

Fig. 19. DimensionsPane Class

Attributes:

- **private final String[] STAR_ICON_LOC:** Locations for star icons.
- **private Button backButton:** The Tutorial Panel will store the back button which will go to main screen.
- **private Button soundButton:** Players can mute the game anytime with soundButton.
- **private Button settingsButton:** settingsButton will go to settings screen.
- **Private Button[] levelButtons:** There are three buttons for three dimensions. All buttons will go to Dimensions screen.
- **private ImageView settingsIcon:** This will store the icon image that displays as a symbol for settings.
- **private ImageView soundIcon:** This will store the icon image that displays as a symbol for sound.
- **private ImageView backIcon:** This will store the icon image that displays as a symbol for back button.
- **Private Label[] leftAt:** There are three labels for three dimensions. These labels will show the levels that the player lastly played.
- **Private Label[] numberOfStarsLabel:** There are three labels for three dimensions. These labels will show the number of stars that the player gained for each dimension.
- **private ImageView[] stars:** There are three images for three dimensions. There will be three star icons which represents how much star does the player has, fullStarIcon, halfStarIcon and emptyStarIcon.
- **private ProgressBar[] dimensionProgression:** There are three progression bar for three dimensions. These progression bars will show the progress of the dimension.
- **Private VBox[] vBox:** VBox for organizing the vertical components of each dimension.
- **Private HBox[] hBox:** HBox for organizing the horizontal components of all dimensions.
- **private final int ICON_SIZE:** Size of the icons.
- **private String iconLocs:** Location of the icons.

Operations:

- **public DimensionsPane():** Constructor of the DimensionPane.
- **public void setNumberOfLevelsLeft(int[] levelsLeft):** Sets the number of levels left for each dimension.
- **public void setNoOfStarsGained(int[] startsGained):** Sets the number of stars gained per dimension.
- **public void setPercentageOfCompletition(int[] percentage):** Sets the percentage of the completion on a given dimension.

4.4.2. Game Mechanics Layer

The Game Mechanics layer basically consist of the classes needed for the gameplay to work. The classes in this layer act as the brain of the game, especially game manager class which controls the other classes in this particular layer.

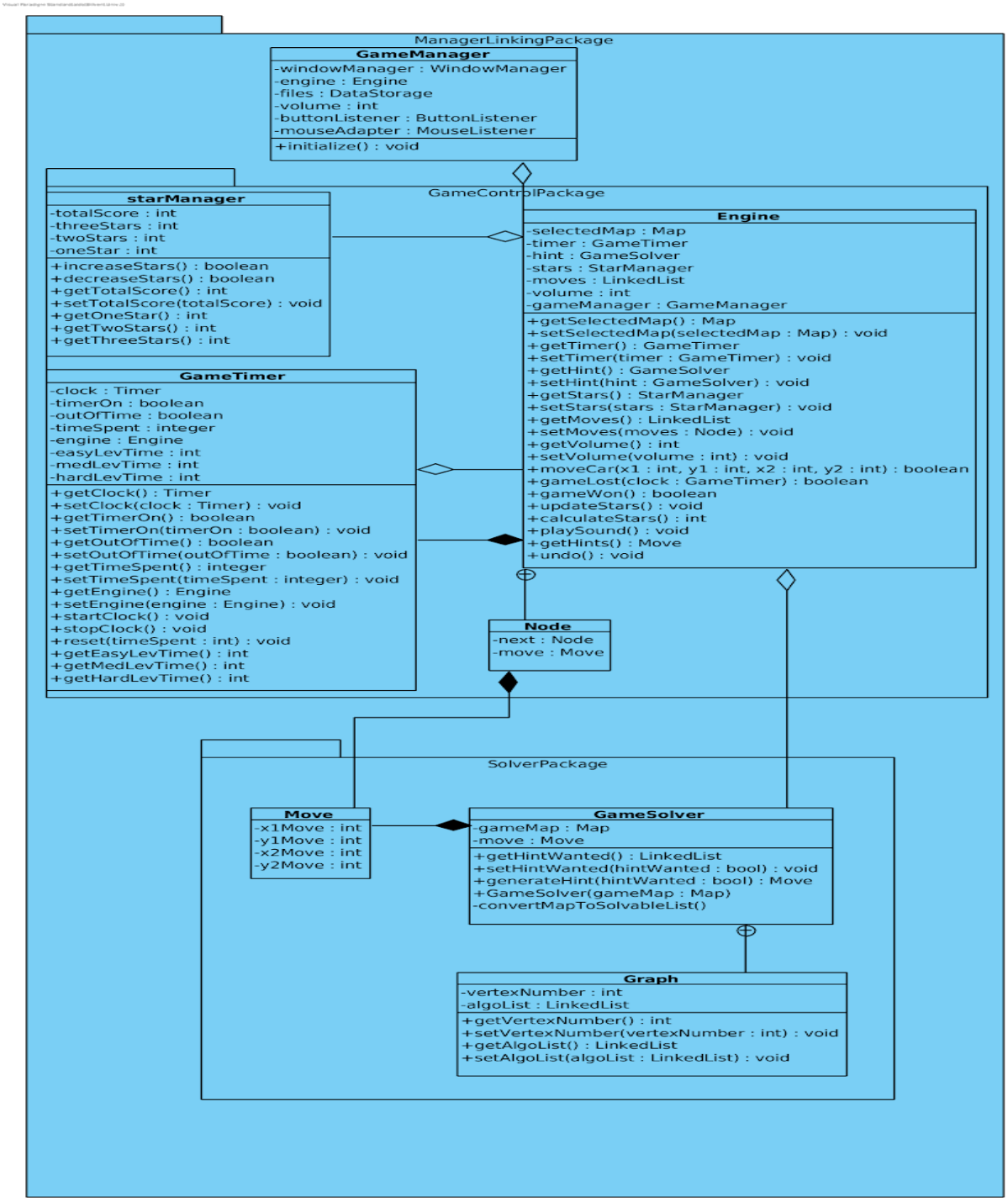

Fig. 20. Game Mechanics Layer Classes

GameManager:

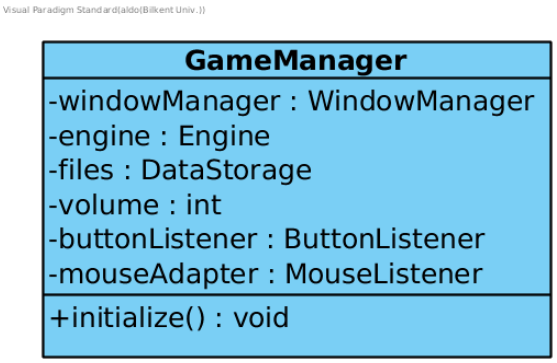

Fig. 21. GameManager Class

The GameManager class is the main class of the program. It will be the class that initializes the UI and creates the connection between the view, the model and the controller.

Attributes:

- **private WindowManager windowManage**r: The GameManager will contain an instance of WindowManger which controls the View of the game.
- **private Engine engine:** engine will be used to make the control the game mechanics and it will be used to create the view of the game.
- **private DataStorage files:** An instance of the files class will be needed in order to store the information of the player when the game is closed. This object will be created the first time the game is opened and loaded when the game is opened again.
- **private int volume:** The volume of the sound effects of the game. The motivation behind saving the volume in this class is because the SettingsPanel must be able to change the volume and the Engine class has to know the current volume level in order to play the sounds.
- **private ButtonListener buttonListener**: The GameManager will contain an instance of the ButtonListener class which will then be added to all the buttons in the screen that is currently viewed by the designed method in the classes extending JPanel.
- **private MouseListener mouseAdapter:** The purpose of this class is similar to the ButtonListener class, except that it will only be used in the panel displaying the grid of the game to make the movement of the cars possible.

Operations:

- **public void initialize():** This is the method that initializes all the properties of the class and creates the connection between them.

Engine:

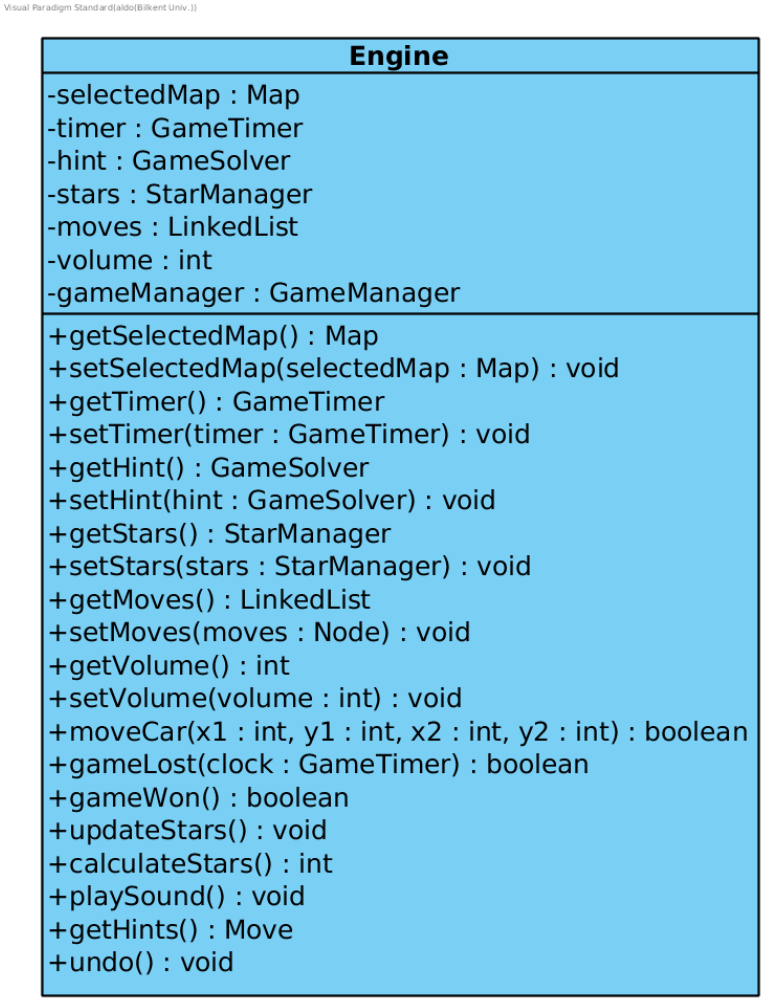

Fig. 22. Engine Class

Attributes:

- **private Map selectedMap:** The map that the player is currently playing.
- **private GameTimer timer:** To track the timer through the game if timer mode is on.
- **private GameSolver hint:** Hint will be generating if the player presses hint button.
- **private StarManager:** To increase or decrease the stars according to the players actions.
- **private LinkedList moves:** The player's moves will be saved as a linked list to use them in undo button.
- **private int volume:** Current sound volume for sound effects.

Operations:

- **public Map getSelectedMap():** Returns selected map to make changes on it.

- **public void setSelectedMap(Map selectedMap):** Selects the map among maps array in the file manager.
- **public GameTimer getTimer():** Returns the timer for GameManager
- **public void setTimer(GameTimer timer):** Sets the timer.
- **public GameSolver getHint():** Returns the gameSolver object.
- **public void setHint(GameSolver hint):** The setter method for the GameSolver object.
- **public StarManager getStars():** Returns the number of stars that the players owns.
- **public void setStars(StarManager stars):** Sets star for selectedMap.
- **public LinkedList getMoves():** Returns the dimensions of the moves that player did in the current game.
- **public void setMoves(LinkedList moves):** Setter method for the moves linked list.
- **public int getVolume():** Returns the current volume.
- **public void setVolume(int volume):** Changes the current volume. This will be saved in the dataStorage later.
- **public boolean moveCar(int x1, int y1, int x2, int y2):** Engine will move the car in the selected map.
- **public boolean gameLost(Timer clock):** This will check If the player is out of time and if yes, will finish the game.
- **public boolean gameWon():** Returns if the game is won.
- **public void updateStars():** Increase or decrease the number of stars in the star manager according to the actions of the player.
- **public int calculateStars():** This method will calculate the star count from the selected map according to optionalMoves in the map and moveCount in the car class.
- **public void playSound(SoundEffect s):** Plays a chosen sound effect.
- **public int[2] getHints():** GameSolver will generate the most optimized move for player to go to exit as an array (x and y dimension), and this function will return that array.
- **public boolean undo():** This method will get the dimensions of the car before the last move and will move the car into that dimensions.

GameTimer:

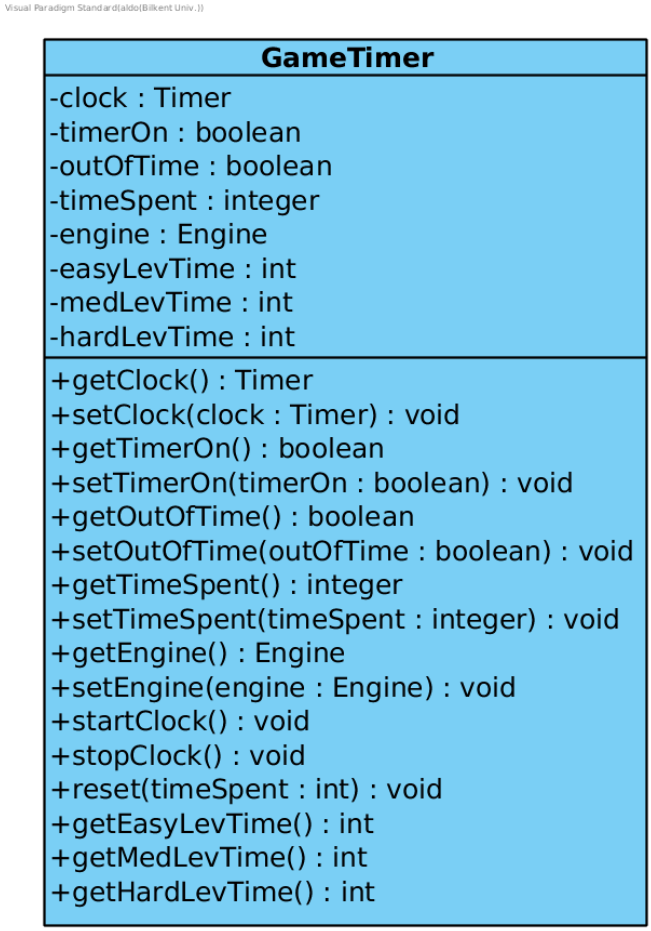

Fig. 23. GameTimer Class

Attributes:

- **private Timer clock:** The java.util class to be used as a timer for the game.
- **private boolean timerOn:** Needed to check if the timer option is enabled or disabled in the game.
- **private boolean outOfTime:** Boolean value to check when the player has run out of time.
- **private int timeSpent:** The time spent by the player represented as the number of seconds.
- **private Engine engine:** A reference to the Engine object that contains this GameTimer object. This reference is needed to stop the game when the player runs out of time.
- **private int easyLevTime:** The time allowed for the easy levels.
- **private int medLevTime:** The time allowed for the medium levels.
- **private int hardLevTime:** The time allowed for the hard levels.

Operations:

- **public Timer getClock():** The getter for the Timer object of the class.
- **public void setClock(Timer clock):** The setter for the Timer object of the class.
- **public boolean getTimerOn():** Checks if the game is in timer mode.
- **public void setTimerOn(boolean timerOn):** Needed to set the game in timer mode.
- **public boolean getOutOfTime():** Checks if the player has run out of time.
- **public void setOutOfTime(boolean outOfTime):** The setter method for the outOfTime property.
- **public int getTimeSpent():** Returns the time spent which will be used to display it for the user.
- **public void setTimeSpent(int timeSpent):** The setter method for the timeSpent property.
- **public Engine getEngine():** Returns a reference to the Engine object of this class.
- **public void setEngine(Engine engine):** Sets the value of the Engine object.
- **public void startClock():** A method needed to start the Timer object of this class.
- **public void stopClock():** Stops the clock when the player loses the game.
- **public void reset():** Resets the timer when the player loses the game and starts the timer mode again.
- **public int getEasyLevelTime():** returns the time needed for the easy mode levels.
- **public int getMedLevelTime():** returns the time needed for the medium mode levels.
- **public int getHardLevelTime():** returns the time needed for the hard mode levels.

GameSolver:

The GameSolver class and its inner Graph class will be used to solve a certain configuration of the game, which will then be used in order to display hints to the user.

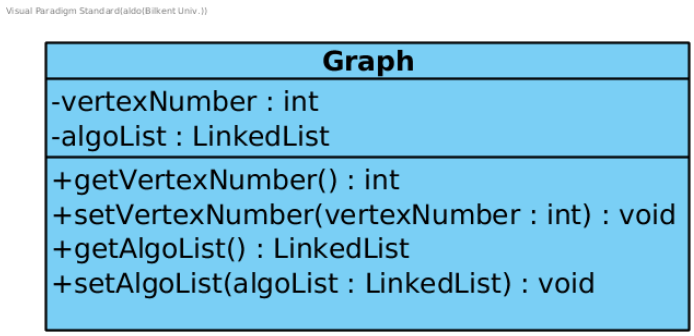

Fig. 24. Graph Inner Class

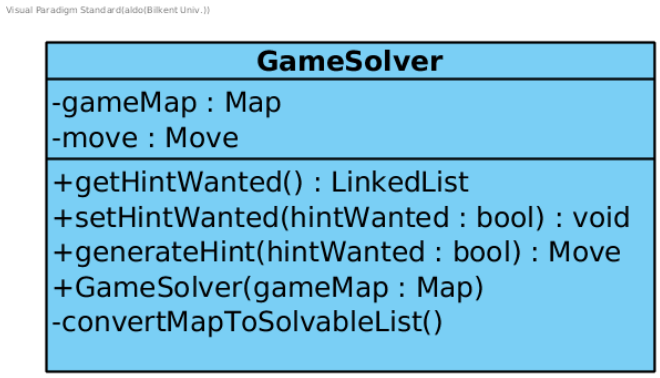

Fig. 25. GameSolver Class and Graph

Attributes:

- **private bool hintWanted:** This property is used to store whether the user has requested a hint or not.

Operations:

- **public bool getHintWanted():** Getter method for the hintWanted property.
- **public void setHintWanted(bool hintWanted):** Sets the value of the hintWanted property.
- **public int[2] generateHint(bool hintWanted):** Generates the x and y coordinates of the hint and returns them to the Engine class which will use this method when the hints are requested by the user.

StarManager:

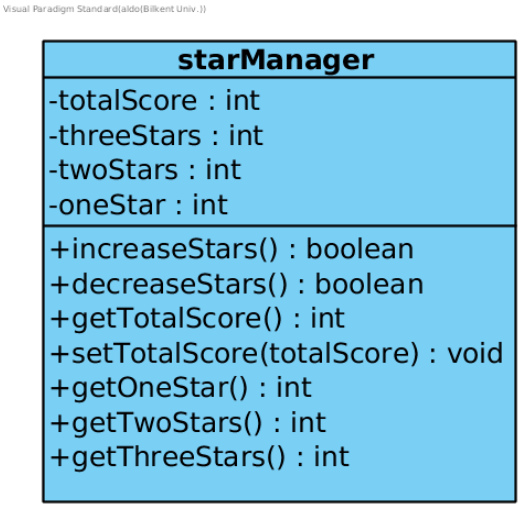

Fig. 26. StarManager Class

Attributes:

- **private int totalScore:** Number of stars that the player owns.
- **private final int THREE_STARS:** For the skins that can be bought for 3 stars and for the maps.
- **private final int TWO_STARS:** For the skins that can be bought for 2 stars and for the maps.
- **private final int ONE_STAR:** For the skins that can be bought for 1 stars and hints and for the maps.

Operations:

- **public boolean increaseStars():** This will increase the totalStars according to the stars that the player gets from selectedMap.
- **public boolean decreaseStars():** This will decrease the totalStars according to the actions of the player. (The player might get an skin or use his/her stars to get a hint etc.)
- **public int getTotalScore():** Returns the totalScore.
- **public void setTotalScore(int totalScore):** The setter method for totalScore property.
- **public int getOneStar():** Returns ONE_STAR value.
- **public int getTwoStar():** Returns TWO_STAR value.
- **public int getThreeStar():** Returns THREE_STAR value.

Node:

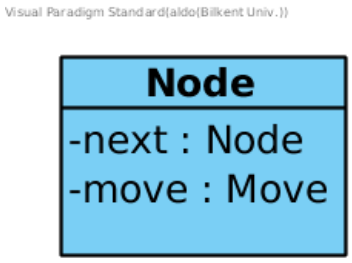

Fig. 27. Graph LinkedList Node Class

Attributes:

- **private next Node:** Holds the reference to the next node of the list.
- **private move Move:** Holds the reference to the move to be made.

Move:

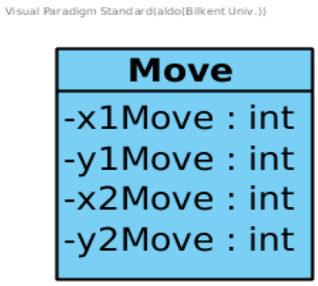

Fig. 28. Move Class

Attributes:

- **private x1Move int:** Defines x start position of the move.
- **private y1Move int:** Defines y start position of the move.
- **private x2Move int:** Defines x end position of the move.
- **private y2Move int:** Defines y end position of the move.

4.4.3. Storage Layer

The figure below represents the Storage Package (storage Layer) of our game which is composed of two sub-packages: FunctionalitiesStorage and GameStorage.

Why we chose the local file system (DataStorage) instead of a database?

The game system proposed by Royal Flush takes into consideration that there is a limited number of files that need to be stored on the system. The files needed for Sound Effects (only 3, one for the game win, one for the game loss and the last for in-game effects), for Settings (only 5 themes), for the Maps (50 maps) and the Cars (13 different cars) is a finite number which is relatively small both when projecting the file sizes as well as when projecting the number of operations needed to retrieve, manipulate and integrate the files into our game. Since all these files are pertaining only to their respective modules, it is easy to directly access them on the local file system. If we were to use a database, however, we would be adding overhead to a job that is as simple as to be solved by accessing the local file system. This discussion does not involve runtimes as the database would be small and fast regardless if we were to keep an index or not to the files. Also considering that the game was designed to fit into a “closed box” type of model, meaning that the user himself cannot in any way add more of these files into the game, the use of a database is clearly an overkill for this design. All this taken into account the group preferred to leave the usage of a database out and deal only with the local file system.

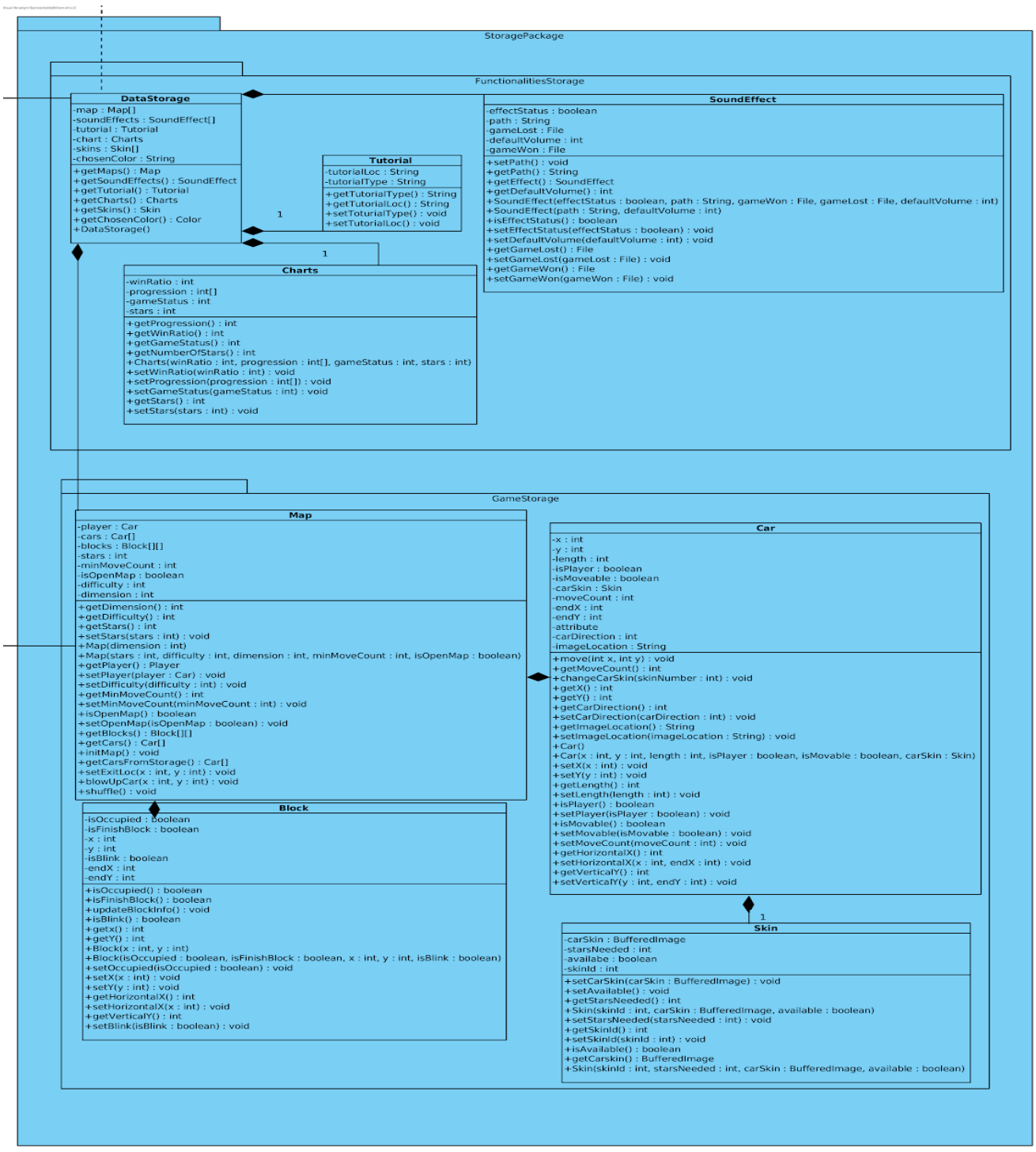

Fig. 29. Storage Layer Classes

Each of the classes included in the storage layer will implement the java.io.Serializable interface as they will be stored after the game is closed. This detail is not included in the class diagram to make it clearer.

DataStorage:

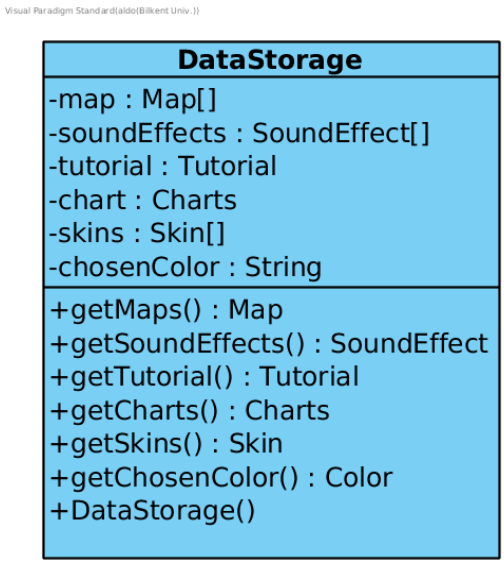

Fig. 30. DataStorage Class

Attributes:

- **private Map maps[]**: Data Storage will store all the maps as an array.
- **private SoundEffect soundEffects[]**: Data Storage will store soundEffects as an array.
- **private Tutorial tutorial:** Data Storage will store the tutorial.
- **private Charts charts:** Data Storage will store the charts.
- **private Skin skins[]:** Data Storage will store the skins.
- **private chosenColor:** Game will have a theme color option and chosen theme will be stored in data storage.

Operations:

- **public DataStorage():** this corresponds to the constructor of the DataStorage.
- **public Map[] getMaps():** Returns the array of maps.
- **public SoundEffect[] getSoundEffects():** Returns the sound effects.
- **public Tutorial getTutorial():** Returns the tutorial.
- **public Charts getCharts():** Returns the charts.
- **public Skin[] getSkins():** Returns the skins.
- **public Color getChosenColor():** Returns the chosen color.

Charts:

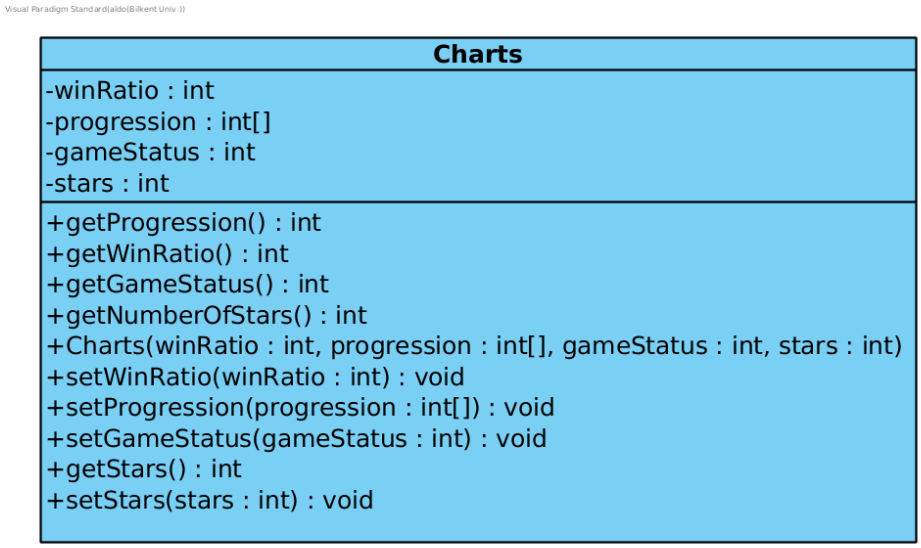

Fig. 31. Charts Class

Attributes:

- **private int winRatio:** Charts will keep win and lose ratio of maps completed in timer mode.
- **private int progression[]:** Progression will keep how much levels they completed in each dimension.
- **private int gameStatus:** Charts will keep general game completion.
- **private int stars:** Charts will keep stars earned through the game.

Operations:

- **public Charts(int winRatio, int progression[], int gameStatus, int stars):** the constructor for the charts class. It initializes its private attributes according to the provided parameters.
- **public int getWinRatio():** Returns the winRatio throughout the game.
- **public int[] getProgression():** Returns the progression throughout the game.
- **public int getGameStatus():** Returns the gameStatus.
- **public int getNumberOfStars()**: Returns the number of collected stars.
- **public void setWinRatio(int winRatio)**: updates the win-lose ratio of the game.
- **public void setProgression(int progression[])**: updates the progression of the game.
- **public void setGameStatus(int gameStatus)**: updates the game status.
- **public int getStars()**: returns the number of collected stars.
- **public int setStars(int stars)**: updates the number of stars.

Tutorial:

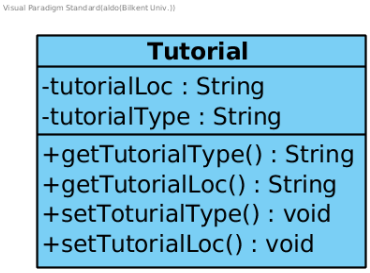

Fig. 32. Tutorial Class

Attributes:

- **private String tutorialLoc:** Location of the tutorial as a string.
- **private String tutorialType**: Type of the tutorial like image or video.

Operations:

- **public string getTutorialLoc ()**: Returns the location of the tutorial.
- **Public string getTutorialType ()**: Returns the type of the tutorial.
- **public void setTutorialLoc (String tutorialLoc):** Sets the location of the tutorial.
- **public void setTutorialType (String tutorialType):** Sets the type of tutorial.

SoundEffect:

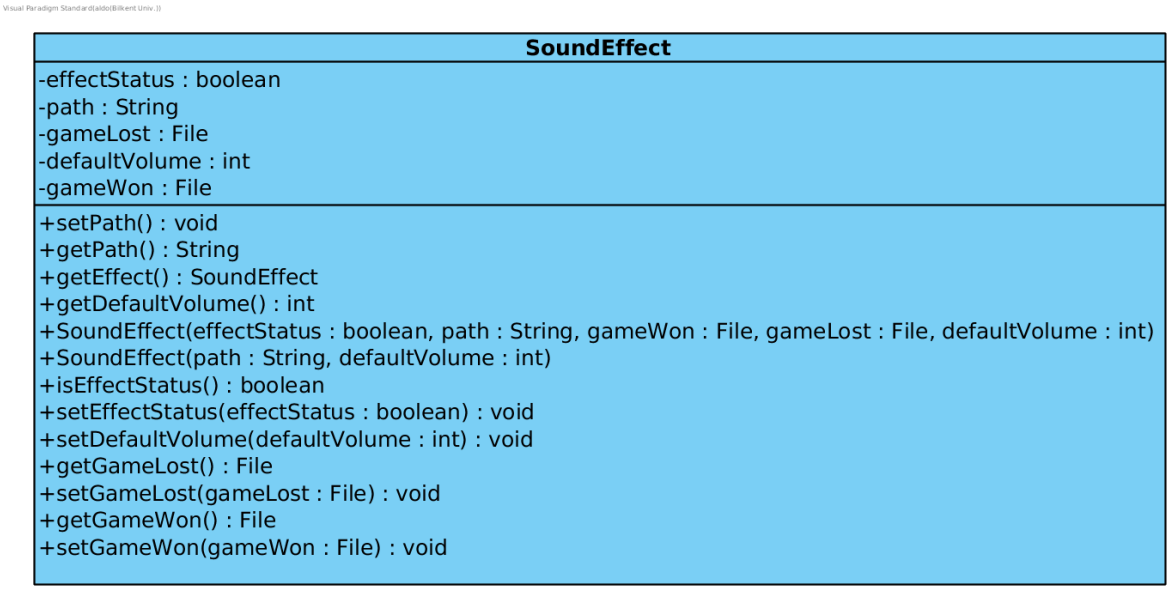

Fig. 34. SoundEffect Class

Attributes:

- **private boolean effectStatus:** Is sound effect on or off.
- **private String path:** Url of the sound effect.
- **private int defaultVolume:** Default volume of the sound effect.
- **private Media gameLost:** The media used when the player loses the game.
- **private Media gameWon:** The media used when the player wins the game.

Operations:

- **Public SoundEffect(String path, int defaultVolume):** constructor that initializes the sound effects to the given parameters. Anything else will be put to a default value.
- **Public SoundEffect(boolean effectStatus, String path, Media gameWon, Media gameLost, int defaultVolume):** A more complete constructor to explicitly determine the attribute values of the classes.
- **public String getPath():** returns the location of the sound.
- **public void setPath(String path):** updates the location of the sound effects.
- **public SoundEffect getEffect():** Returns the sound effect.
- **public int getDefaultVolume():** Returns the default volume.
- **public boolean isEffectStatus():** returns 'True' only for the chosen soundEffect.
- **public void setEffectStatus(boolean effectStatus):** makes one of the sound effects the chosen sound effect by making its soundEffect attribute 'True'.
- **public void setDefaultVolume(int defaultVolume):**
- **public Media getGameLost():** returns the game lost media.
- **public void setGameLost(Media gameLost):** updates the game lost media.
- **public Media getGameWon():** returns the game win media.
- **public void setGameWon(Media gameWon):** updates the game win media.

Map:

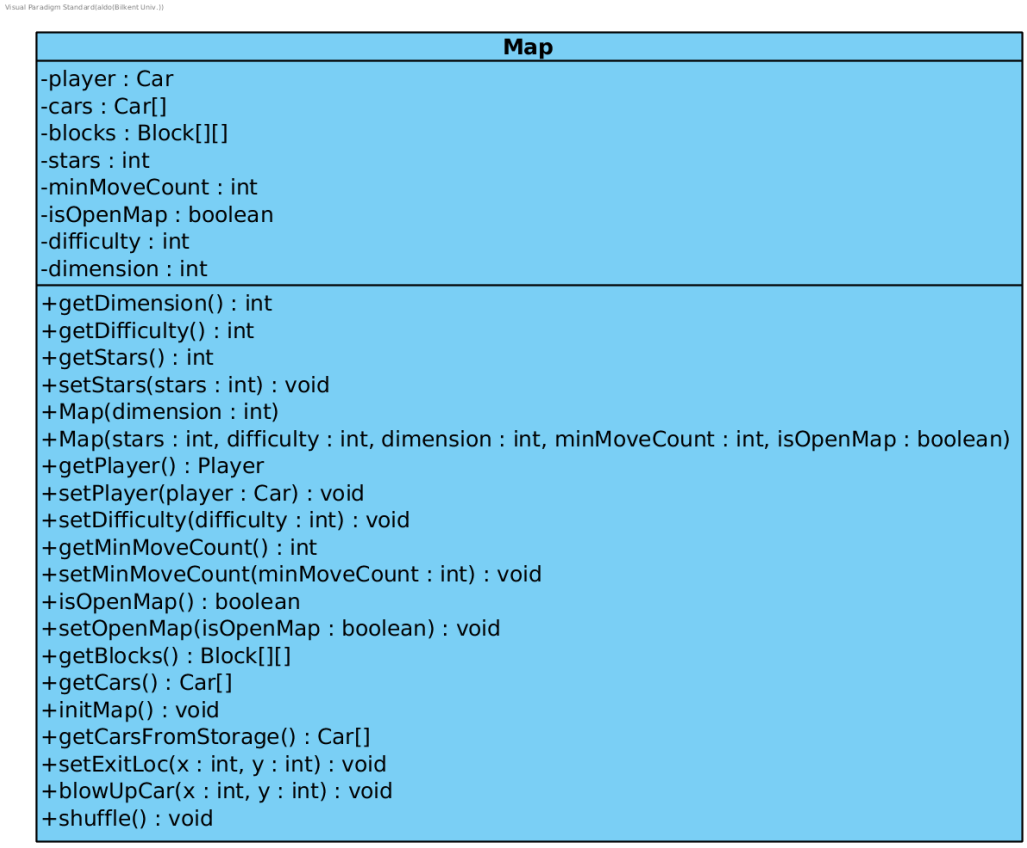

Fig. 33. Map Class

Attributes:

- **private Car player:** Car that tries to leave the park.
- **private Car cars[]:** Movable cars that block player.
- **private Block blocks[][]:** Two-dimensional blocks that create the map.
- **private int stars:** Number of stars earned from that map according to moveCount of the player and optimalMoveCount.
- **private int minMoveCount:** Optimal move counts to complete the map. This will be used to give stars up to 3.
- **private Boolean isOpenMap:** True if map is accessible for the player.
- **private int difficulty:** used to determine the difficulty of the map.
- **private int dimension:** the dimensions of the square grid of the map.

Operations:

- **Public Map(int dimension):** constructor for the map, the other properties get set to 0.
- **Public Map(int stars, int difficulty, int dimension, int minMoveCount, boolean isOpenMap):** complete constructor for the map class. Sets all the private attributes to the initial values of the parameters.
- **public int getDimension()**: Returns the dimension of the map according to number of blocks in the map.
- **public int getDifficulty():** Returns the difficulty of the map.
- **public void setDifficulty(int difficulty):** Updates the difficulty of the map.
- **public int getStars():** Returns earned stars.
- **public int setStars(int stars):** Updates earned/collected stars.
- **public Car getPlayer():** returns the player of the map.
- **public void setPlayer(Car player):** updates which car in the map behaves as the player.
- **public int getMinMoveCount():** returns the minimal move count.
- **public void setMinMoveCount(int minMoveCount):** updates the minimal move count.
- **public boolean IsOpenMap():** returns true only if the current map is opened in playmode.
- **public void setOpenMap(boolean isOpenMap):** update the information for open map.
- **public Block[][] getBlocks():** returns the two-dimensional array of the map blocks.
- **public Car[] getCars():** return the cars in the map.
- **Init void initMap():** initialize the map to the default values.
- **public void addCar():** This will add cars to the map.
- **public int isMapFinished():** Checks if the player is in the end block and label the map as finished. This will give the stars and update the charts.

Car:

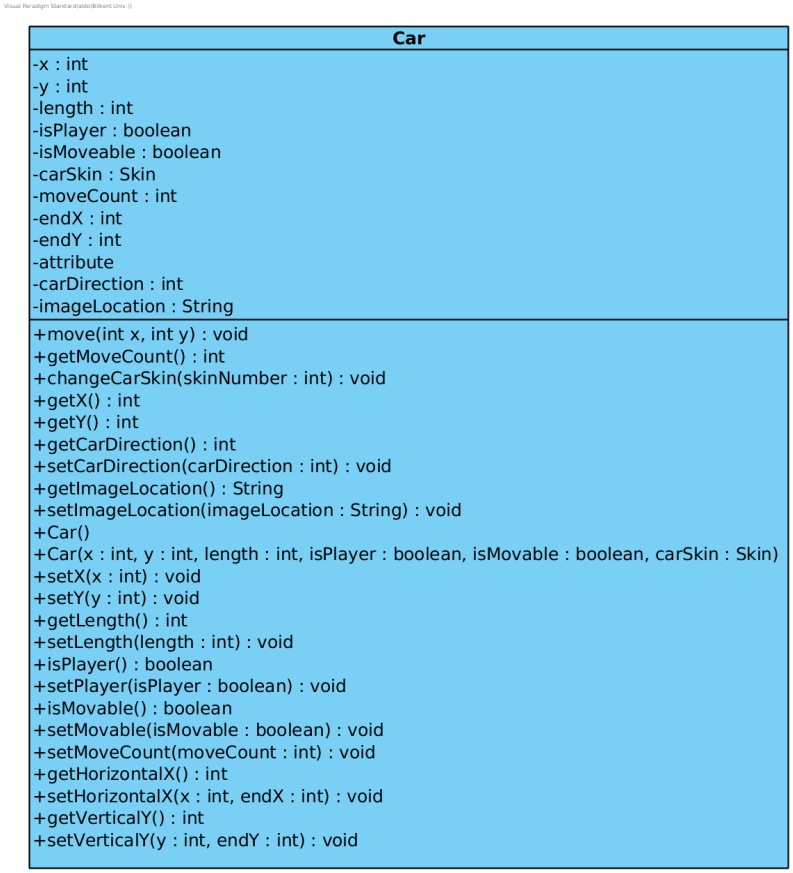

Fig. 35. Car Class

Attributes:

- **private int x**: X of the car.
- **private int y:** Y of the car.
- **private int length:** Length of the car (could be 2 or 3).
- **private boolean isPlayer:** Car could be a player or obstacle.
- **private boolean isMoveable:** Car could be movable or fixed.
- **private Skin carSkin:** Car will have a default skin but it can be changed later.
- **private int moveCount:** Move count of the player.
- **private int endX:** the horizontal end position of the car.
- **private int endY:** the vertical end position of the car
- **private int carDirection:** can be one of the 1,2,3 or 4 corresponding to left, up, right and down.
- **private String imageLocation:** the location of the image of the car.

Operations:

- **public Car():** constructor for the Car.
- **public Car(int x, int y, int length, boolean isPlayer, boolean isMovable, Skin carSkin):** a more comprehensive constructor that sets all the attributes to the given parameter values.
- **public void move():** If the next block is not occupied, car can move either in x direction or y-direction.
- **public int getMoveCount()**: Returns move count of the player for each map at the end of the level.
- **public void changeCarSkin(int skinNumber):** Change the car skin with earned stars.
- **public void setX(int x):** updates the initial x position.
- **public int getX():** Returns the horizontal dimension of the car.
- **public void setY(int y):** updates the initial y position.
- **public int getY():** Returns the Y dimension of the car.
- **public int getCarDirection():** returns the direction of the car.
- **public void setCarDirection(int carDirection):** updates the orientation of the car.
- **public String getImageLocation():** returns the image location of the car.
- **public void setImageLocation(String imageLocation):** updates the image location of the car.
- **public int getLength():** returns the length of the car.
- **public void setLength(int length):** updates the length of the car.
- **public boolean isPlayer():** returns true if it is the car of the player.
- **public void setPlayer(boolean isPlayer):** sets the current car as the player.
- **public boolean isMovable():** returns true if the car is allowed to move.
- **public void setMovable(boolean isMovable):** updates the movable attribute of the car.
- **public void setMoveCount(int moveCount):** updates the move count.
- **public int getHorizontalX():** returns the x position of the end of the car.
- **public void setHorizontalX(int x, int endX):** updates both initial and end position of the car in the x-direction.
- **public int getHorizontalY():** returns the y position of the end of the car.
- **public void setHorizontalY(int y, int endY):** updates both initial and end position of the car in the y direction.

Block:

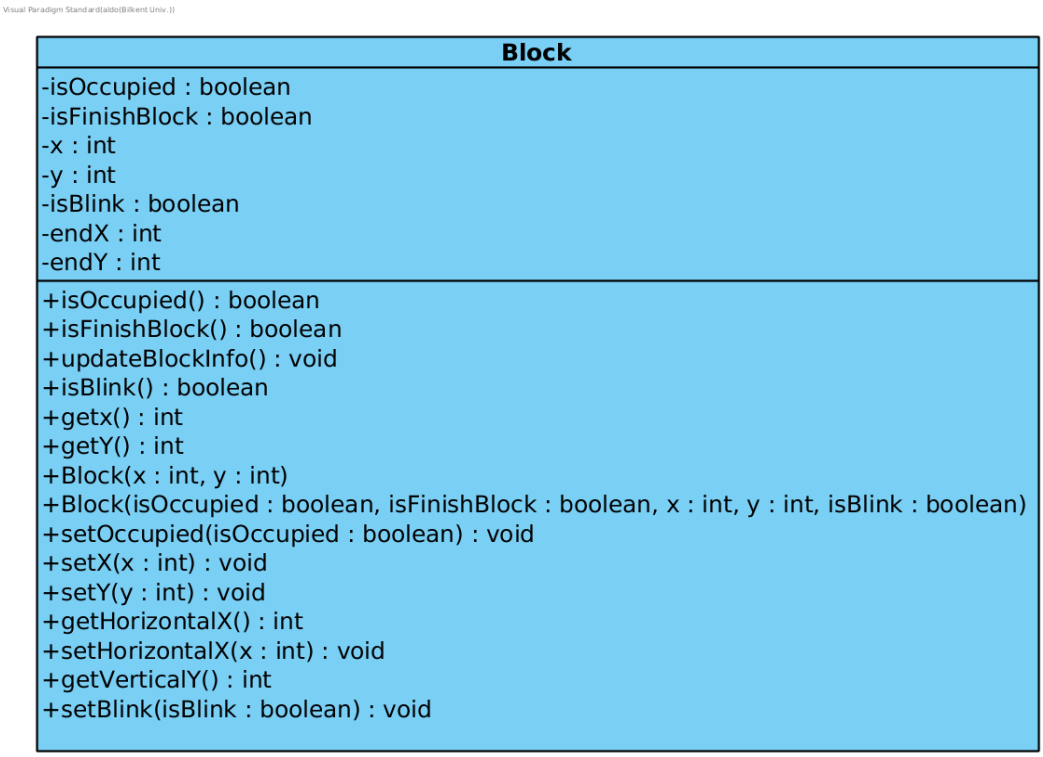

Fig. 36. Block Class

Attributes:

- **private boolean isOccupied:** True if there is a car on the block.
- **private boolean isFinishBlock:** Finish block means the block which ends the game if player is on that block. There will be one finish block for each map.
- **private int x:** X dimension of the block.
- **private int y:** Y dimension of the block.
- **private int isBlink:** During the tutorial and hint, some blocks will blink in order to give clues about the next move.
- **private int endX:** the x dimension of the ending of the block.
- **private int endY:** the y dimension of the ending of the block.

Operations:

- **Public Block(int x, int y):** constructor for the block. Initial locations set to x and y.
- **Public Block(boolean isOccupied, boolean isFinishBlock, int x, int y, boolean isBlink):** a more comprehensive constructor providing for more attributes to the block.

- **Public void setOccupied(boolean setOccupied):** updates the occupied information of the block.
- **public boolean isOccupied()**: Returns isOccupied.
- **public boolean isFinishBlock():** returns true if the block is the exit block.
- **public boolean isFinished():** Returns true if there is a car on the finish block.
- **public void updateBlockInfo()**: Updates block info when a car moves onto the block.
- **public boolean IsBlink():** Returns true if the block is blinking.
- **public int setX():** updates the x position of the block.
- **public int getX():** Returns the X dimension of the block.
- **public int setY():** updates the y position of the block.
- **public int getY():** Returns the Y dimension of the block.
- **public int getHorizontalX():** returns the x position of the block.
- **public void setHorizontalX(int endX):** updates the horizontal x position of the block.
- **public int getVerticalY():** returns the vertical y position of the block.
- **public void setVerticalY(int endY):** updates the end vertical position of y.
- **public void setBlink(boolean isBlink):** sets the blinking property of the block.

Skin:

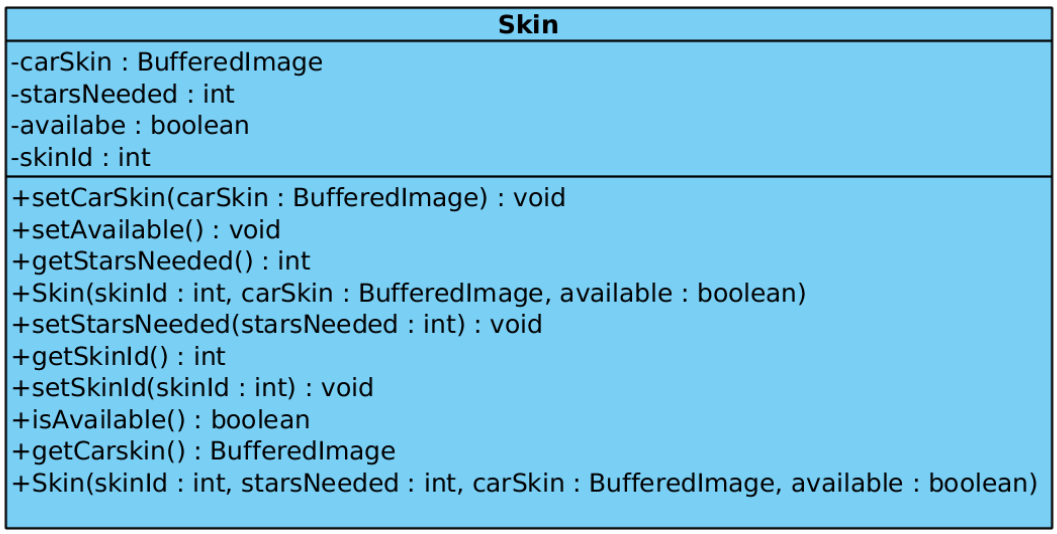

Fig. 37. Skin Class

Attributes:

- **private ImageView carSkin:** Image of the skin.
- **private int starNeeded:** Skins will have some values to buy them.
- **private boolean available:** Some skins will not open if you do not have stars.
- **private int skinID:** a unique identifier for the skin.

Operations:

- **public Skin(int skinID, int starsNeeded, ImageView carSkin, boolean available):** constructor for the skin object.
- **public Skin(int skinID, ImageView carSkin, boolean available):** second constructor for the skin class.
- **public void setCarSkin(ImageView carSkin):** updates the car skin.
- **public void setAvailable(boolean available):** updates the available property of the skin.
- **public int getSkinID():** returns the skin id.
- **public void setSkinID(int skinID):** updates the unique identifier of the skin.
- **public boolean isAvailable()**: Returns true if the skin is open.
- **public void setStarNeeded()**: updates the number of stars needed for this skin.
- **public int getStarNeeded()**: Returns the star needed for opening the skin.
- **public ImageView getCarSkin():** Returns the car skin.

5. Improvement Summary

In the second iteration of the design report, we decided to make several changes in our game.

- There will be some game mechanics changes. We decided to add more options for players to use when they are stuck in the game. There will be an additional panel which is called “OptionsPane” in the PlayGamePane which includes:
    - a “blow-up car” button that disappears one car from the map,
    - a “rotate car” button that rotates one car from its center,
    - a “shrink” button that makes a car one block-size smaller,
    - a “change exit” button to change the exit of the game by one block size,
    - a “hints” button which previously was on the middle right section of the screen.
- We changed our “FileManager” name to “DataStorage”, “Panel” names to “Pane”s, added functions pertaining to the new features that were added to the game and redesigned our diagrams corresponding to these changes.

- Subsystem Decomposition is updated according to sub-compositions of the logic of the game such as models, views and controllers.
- Model Subsystem is changed according to the feedback received in the first iteration demo and to the conventions, we learned in the lectures.
- We deleted our User Experience vs. Memory trade-off since it is more like a boundary condition and edited the Maintainability vs. Space trade-off according to the feedback we've gotten. We also found more trade-offs in our game and added Rapid Development vs. Functionality, Functionality vs. Usability and Efficiency vs. Portability.
- We added a 3-Layer Architectural diagram to further explain our software architecture from the perspective different than the MVC principle.
- We decided to use JavaFX libraries for our GUI, and we changed our User Interface Layer according to our implementation in the first iteration.
- Another added feature of the game is the "Colour Blind" option where the users who are unable to differentiate certain colors will be able to play the game with the help of the cars with symbols which will indicate which car is the player and which is not.
- We created three internal packages according to the main parts of our game, added them to our final object design diagram, and changed *Internal Packages* subtitle in the report. We also changed our external packages since we are using JavaFX in this iteration.
- We also included an access matrix diagram to show which actors are able to access which part of the game in order to prevent security issues.

6. *References*

[1] Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110-0.