Bilkent University

Department of Computer Engineering

CS319 Term Project

Rush Hour Game

Design Report

Aldo Tali

Barış Can

Endi Merkuri

Hygerta Imeri

Sıla İnci

Instructor: Eray Tüzün

Assigned TA: Muhammed Çavuşoğlu

Design Report

November 8, 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 a 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 be also several game themes and car skins to increase variety through the game by using some stars so that the game is felt 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 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 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 the case that 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 local data by using the Java Serializable interface. Therefore, there will not be any security issues. 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

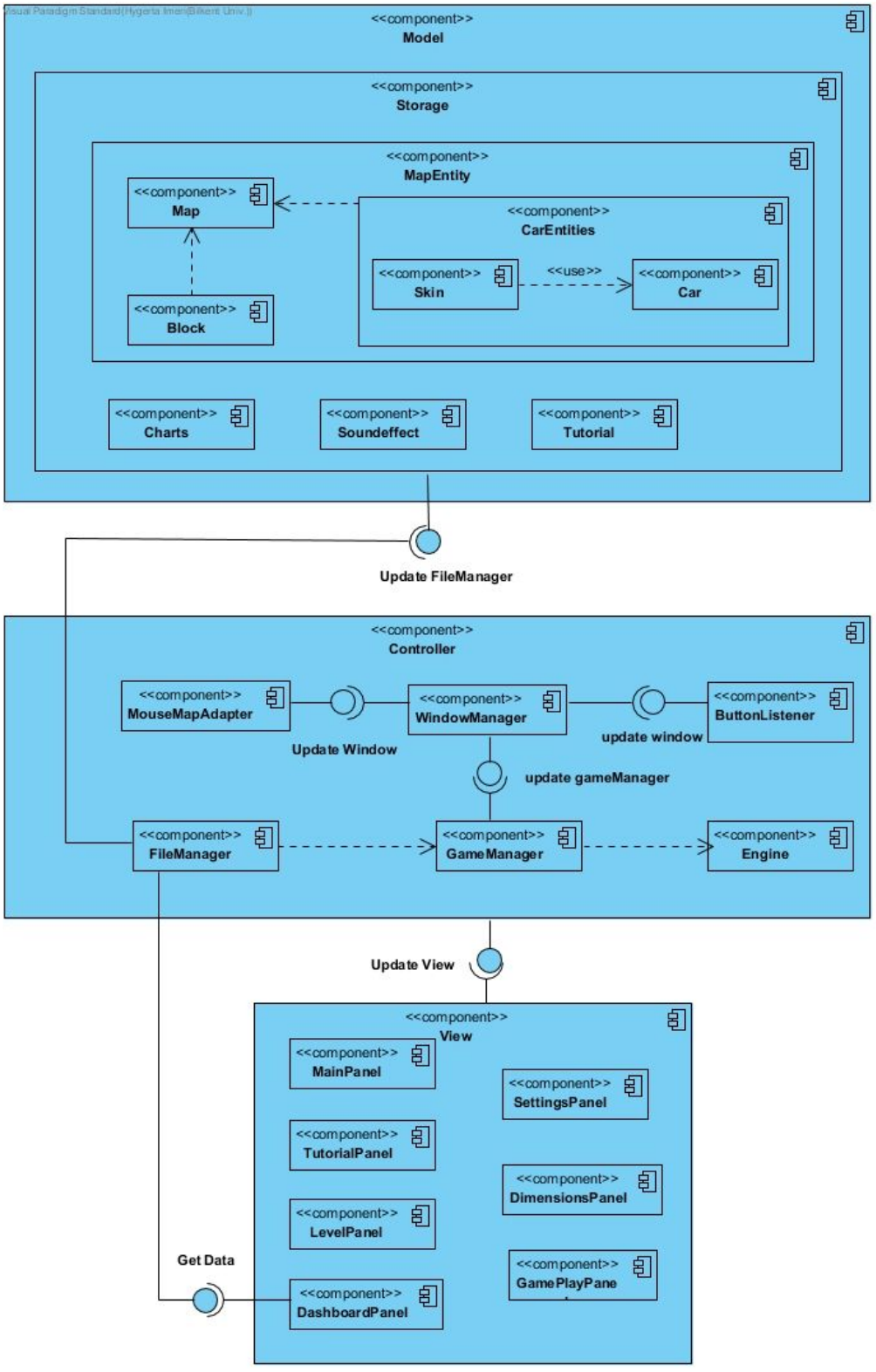
Since there is no internet connection in Rush Hour, portability is very important in order to open your saved games from another computer. The game will have a small size so that it can be easily transferred into another computer easily without any problem. Moreover, because the save files will be kept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

2.1 Subsystem Decomposition



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.

For the design pattern of this project we have chosen to use and apply *Model-View-Controller* (MVC). While designing our component diagram we have divided our subsystems in order for it to be fit 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 the Swing widget toolkit and the 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 game play 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 FileManager. 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 game play 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.

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.

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

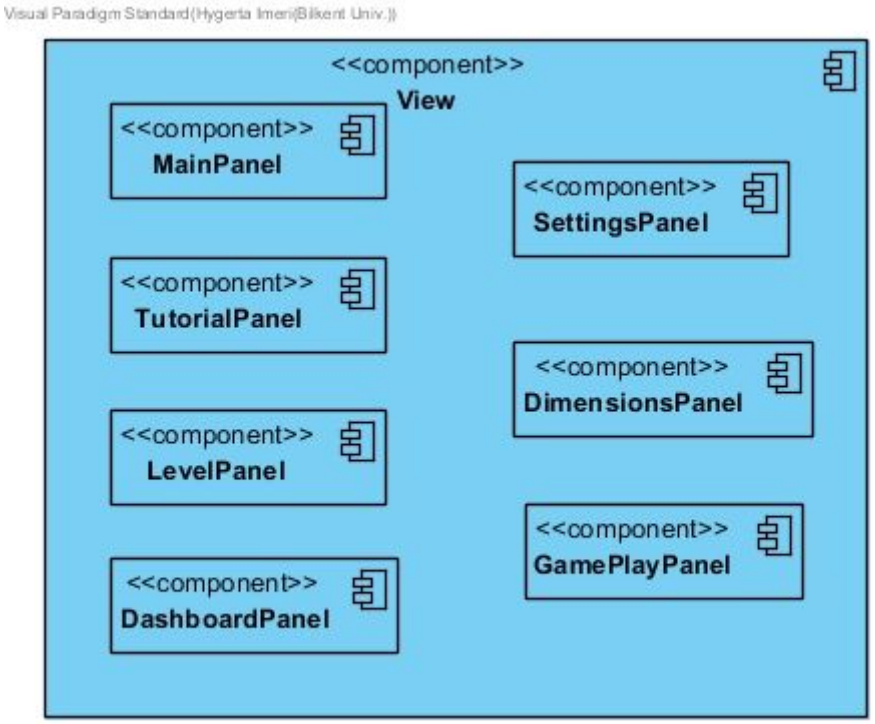
supportability this ads to the portability as well. The executable is easily portable in different devices and as such can be run accordingly.

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 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*

3.1 UI Subsystem



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. MainPanel
2. SettingsPanel
3. TutorialPanel
4. DimensionsPanel
5. LevelPanel
6. GamePlayPanel
7. DashboardPanel

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

The *SettingsPanel* 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 *TutorialPanel* 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.

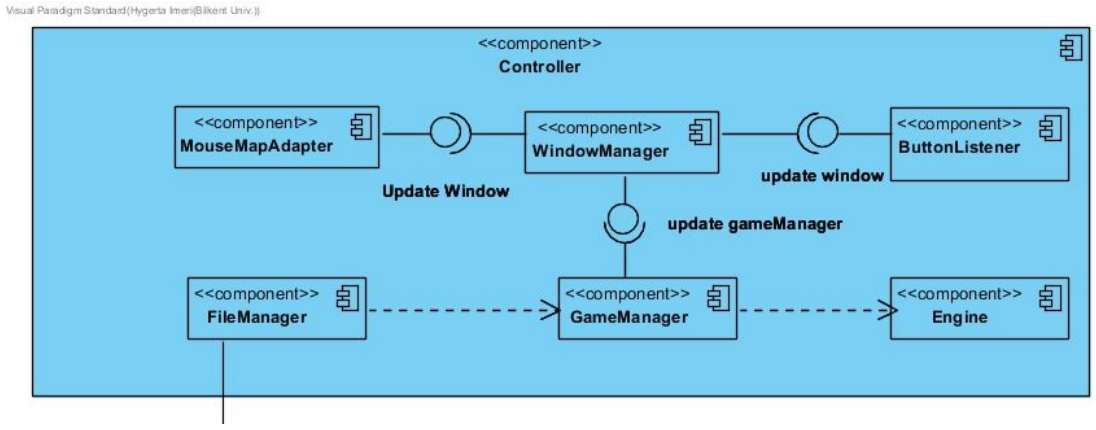
The *DimensionsPanel* view is responsible for displaying dimensions for the maps the user can select. This screen also has smaller buttons for going straight to settings and again, adjusting the volume. The screen also requires the amount of stars from the FileManager in order to show them below the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

The *LevelPanel* 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 *GamePlayPanel* 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 form the GameManager.

The *DashboardPanel* 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 **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. MouseMapAdapter
2. WindowManager
3. ButtonListener
4. FileManager
5. GameManager
6. Engine

The *MouseMapAdapter* provides the information about where the mouse was clicked on the screen to WindowManager. MouseMapAdapter 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 MouseMapAdapter, 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.

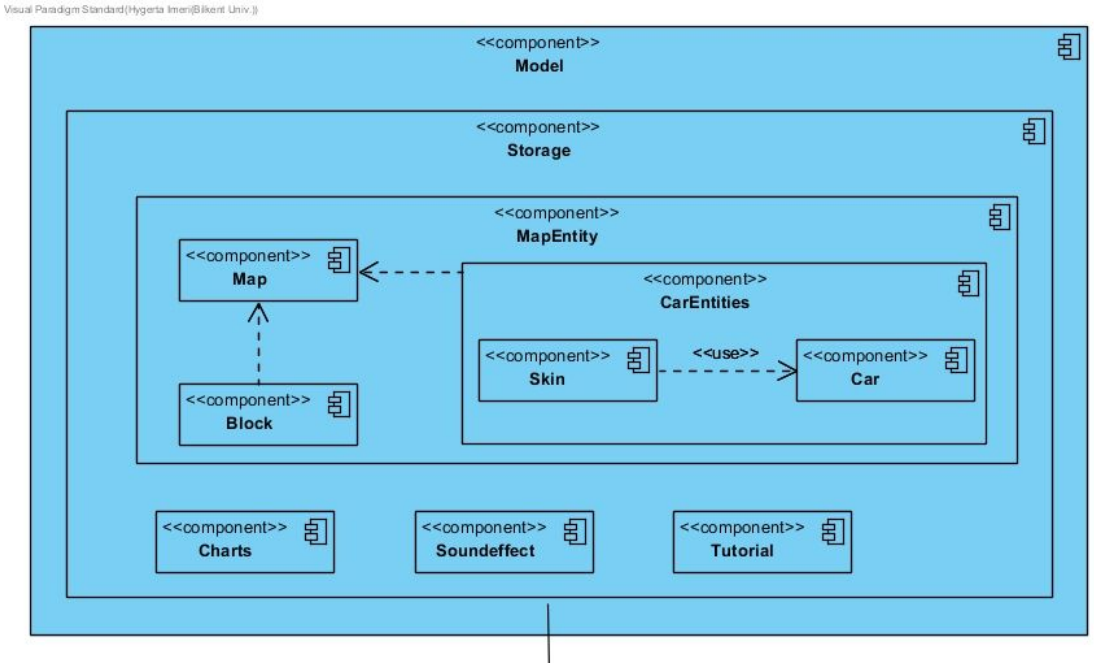
The *FileManager* is responsible for all the storage and managing the operations related to showing the data on the Dashboard screen. It also has a dependency relation with the GameManager regarding that the GameManager controls all the data flow that requires for the game to operate accordingly.

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, soudnd effects etc. from the FileManager. 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 FileManager, Engine will select a map from FileManager 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 **Model Subsystem** contains the classes that are used to store and manipulate some state. This subsystem updates FileManager. 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
    a. MapEntity
        i. Map
        ii. Block
        iii. CarEntities
            - Skin
            - Car

    b. Charts

    c. SoundEffects

    d. 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 FileManager and include their operations as well.

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. Because the overhead in the size of the objects will be very small and the objects that contain these properties will not be stored, the space used will not be problematic.

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.

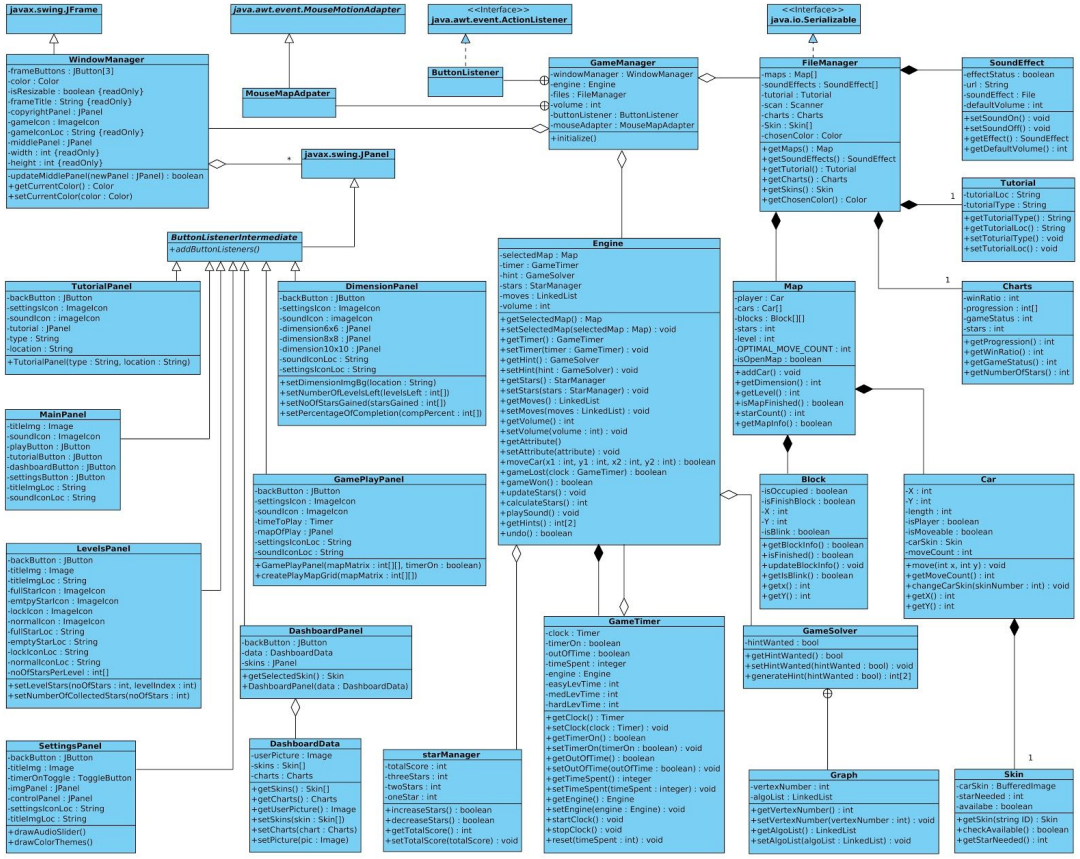
User Experience vs Memory:

As the software being implemented is a game, its main focus is making it more user friendly compared to other more specialized programs. To do this, we have decided to make the user interface simple with not many complex features or options. Since this is the primary goal, it may require relatively more memory than some of the other simple games, but there will be

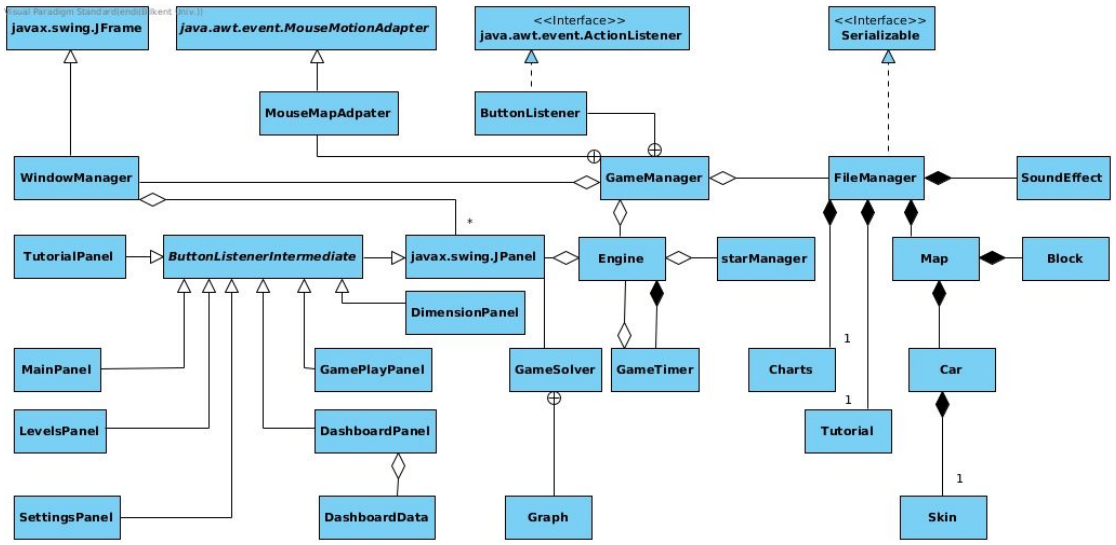
no decrease in performance since the game will still have low hardware requirements because of its simplicity and implementation efficiency.

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. An abstraction of this class diagram containing only the classes and their relations is also provided.



Final Object Design Abstraction:



4.3. Packages

4.3.1. Internal Packages

We have not defined any packages at this stage of the development but this decision is bound to change in later iterations. The best way to create packages would be to divide the classes in a way to represent the MVC design pattern, which may be done corresponding to the subsystem that were presented in the earlier parts of the report.

4.3.2. External Packages (Java packages)

javax.swing

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

java.awt.event

We will use this java package to make the classes created by the javax.swing 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.

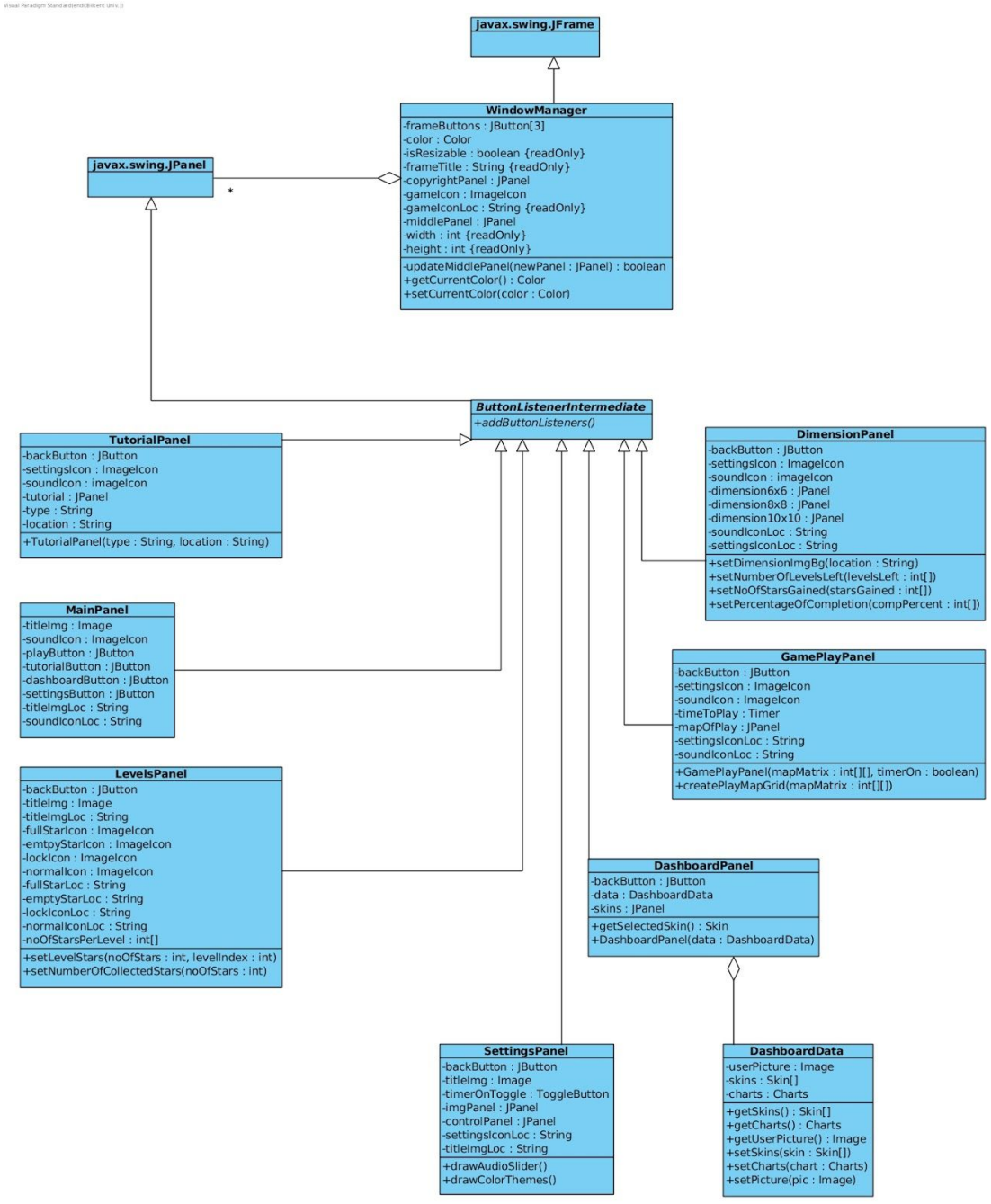
java.util

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.

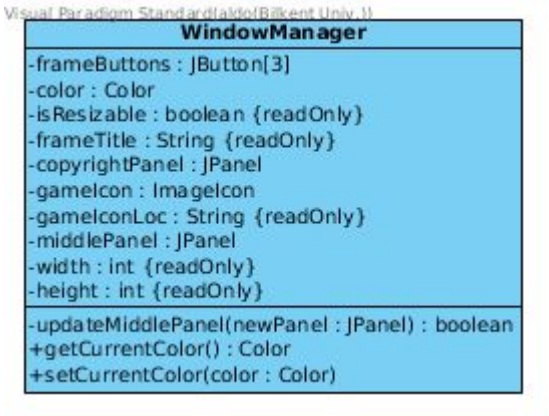
4.4. Class Interfaces

Note: The above class diagram is divided in 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



WindowManager:



Attributes:

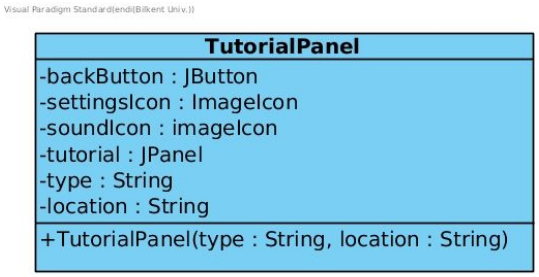
- **private JButton[3] frameButtons:** The WindowManager will keep a Jbutton array of size 3 to keep the minimize, maximize and close buttons.
- **private Color color:** color will keep the theme of the game.
- **private static final boolean isResizable:** isResizable will keep the resizability information of the frame.
- **private static final String frameTitle:** frameTitle will keep the title of the frame in the game.
- **private JPanel copyrightPanel:** copyrightPanel will keep the bottom most Panel of the screen.
- **private ImageIcon gameIcon:** keeps the icon found in the top leftmost corner of the frame.
- **private static final String gameIconLocation:** keeps the location where to read the game Icon from.
- **private JPanel midlePanel:** midle Panel will keep the middle Panel of the screen. This is the panel that keeps changing based on user operations.
- **private static final int width:** stores the width of the game frame.
- **private static final int height:** stores the height of the game frame.

Operations:

- **public boolean updateMiddlePanel(JPanel newPanel):** Updates the central GUI panel to the application.

- **public Color getCurrentColor():** Returns the current selected theme of the user.
- **public void setCurrentColor(Color color):** Sets the current selected theme of the user.

TutorialPanel:



Attributes:

- **private JButton backButton:** The Tutorial Panel will store the back button that it displays on the top leftmost corner.
- **private ImageIcon settingsIcon:** This will store the icon image that displays as a symbol for settings.
- **private ImageIcon soundIcon:** This will store the icon image that displays as a symbol for sound.
- **private Jpanel tutorial:** This will store the tutorial to be displayed on the screen for the gameplay.
- **private String type:** The type of the current tutorial.
- **private String location:** The location of the current tutorial.

Operations:

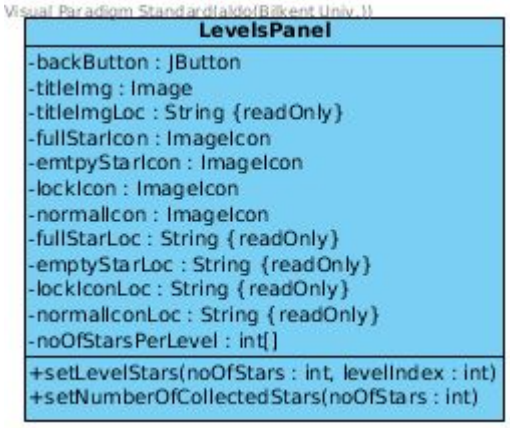
- **public TutorialPanel(type: String, location: String):** This is the constructor for the tutorial panel. It specifies the type of the tutorial and the location from where to get the media files.

MainPanel:

Attributes:

- **private Image titleImg:** This image holds the one that displays as the title on the main screen.
- **private ImageIcon soundIcon:** The image holds the icon that is used for showing the sound.
- **private JButton playButton:** This is the button that leads the player to the Dimensions screen.
- **private JButton tutorialbutton:** This is the button that leads the player to the Tutorials Screen.
- **private JButton dashboardButton:** This is the button that leads the player to the Dashboard Screen.
- **private JButton settingsButton :** This is the button that leads the player to the Settings Screen.
- **private static final String titleImgLoc :** This keeps the location of the title image in the play screen.
- **private static final String soundIconLoc :** This keeps the location of the sound icon in the play screen.

LevelsPanel:



Attributes:

- **private JButton backButton:** The Tutorial Screen will store the back button that it displays on the top leftmost corner.
- **private Image titleImg:** This image holds the one that displays as the title on the main screen.
- **private static final String titleImgLoc :** This stores the location of the image icon in the folder directory.
- **private ImageIcon fullStarIcon:** This is the image of a full star that is used on the levels display for progress.
- **private ImageIcon emptyStarIcon:** This is the image of an empty star that is used on the levels display for progress.
- **private ImageIcon lockIcon:** This is the image of locked key that is used on the levels display for blocked levels.
- **private ImageIcon normalIcon:** This is the image of that is used by default to display unblocked levels in the game.
- **private static final String fullStarLoc :** This stores the location of the image icon needed for the fullStar display.
- **private static final String emptyStarLoc :** This stores the location of the image icon needed for the emptyStar display.
- **private static final String lockIconLoc :** This stores the location of the image icon needed for the blocked level display.

- **private String normalIconLoc :** This stores the location of the image icon needed for the default level display.

Operations:

- **public void setNoOfStarsPerLevel(int index Level, int noStars):** This method updates the number if stars for the level at index indexLevel.
- **public boolean 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.

SettingsPanel:

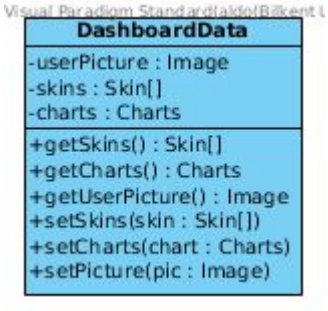
Attributes:

- **private JButton backButton:** The Tutorial Screen will store the back button that it displays on the top leftmost corner.
- **private Image titleImg:** This image holds the one that displays as the title on the main screen.
- **private static final String titleImgLoc :** This stores the location of the image icon in the folder directory.
- **private ToggleButton timerOnToggle:** This stores the toggle button that turns timer on or of.
- **private JPanel imgPanel:** This panel stores the visual section of the Panel.
- **private JPanel controlPanel:** This panel store the user interactive section of the screen.
- **private static final String settingsIconLoc:** This stores the location of the settings icon in the folder directory.

Operations:

- **public void drawAudioSlider():** This method draws the slider needed in the audio volume percentage selection in the settings screen.
- **public void drawColorThemes():** This method draws the color theme selection in the settings screen.

DashboardPanel:



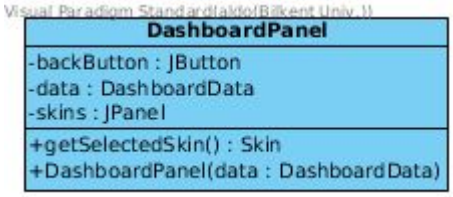
Attributes:

- **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 to be drawn on the dashboard screen.

Operations:

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

DashboardPanel:



Attributes:

- **private JButton backButton:** The Tutorial Screen will store the back button that it displays on the top leftmost corner.
- **private DashboardData data:** This holds all the possible displayable data in the Dashboard.
- **private JPanel skins:** This holds the panel of selectable skins.

Operations:

- **public Skin getSelectableSkin():** Returns the current selected skin to be displayed to the user.
- **public Dashboard(DashboardData data):** This serves as the constructor of the dashboard.

GamePlayPanel:

Attributes:

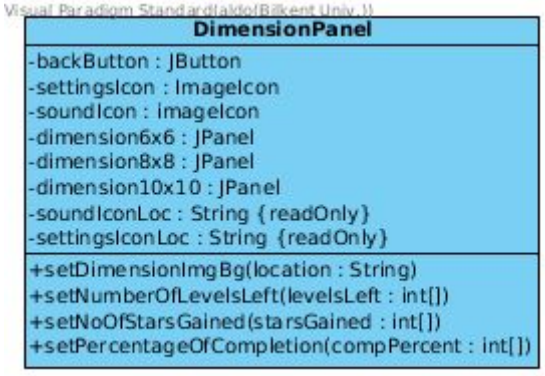
- **private JButton backButton:** The Tutorial Screen will store the back button that it displays on the top leftmost corner.

- **private ImageIcon settingsIcon:** This will store the icon image that displays as a symbol for settings.
- **private ImageIcon soundIcon:** This will store the icon image that displays as a symbol for sound.
- **private Timer timeToPlay:** Keeps the timer that is restricted on the game play. The timer is in a countdown fashion.
- **private static final String settingsIconLoc:** This stores the location of the settings icon in the folder directory.
- **private static final String soundIconLoc:** This stores the location of the sound icon in the folder directory.
- **private JPanel mapOfPlay:** This keeps the playGame panel with the map grid and the moving car

Operations:

- **public void createPlayMapGrid(mapMatrix int[][]):** Creates the playable grid in the GamePlayPanel.
- **public GamePlayPanel(mapMatrix int[][], timerOn boolean):** This serves as the constructor of the GamePlayPanel.

DimensionPanel:



Attributes:

- **private JButton backButton:** The Tutorial Screen will store the back button that it displays on the top leftmost corner.
- **private ImageIcon settingsIcon:** This will store the icon image that displays as a symbol for settings.

- **private ImageIcon soundIcon:** This will store the icon image that displays as a symbol for sound.
- **private JPanel dimension6x6:** This will store the 6x6 dimension clickable display.
- **private JPanel dimension8x8:** This will store the 8x8 dimension clickable display.
- **private JPanel dimension10x10:** This will store the 10x10 dimension clickable display.

Operations:

- **public boolean setDimentionImgBg(String location):** Sets the location of the dimension image background.
- **public boolean setNumberOfLevelsLeft(int[] levelsLeft):** Sets the number of levels left for each dimension.
- **public boolean setNoOfStarsGained(int[] startsGained):** Sets the number of stars gained per dimension.
- **public boolean setPercentageOfCompletition(int[] percentage):** Sets the percentage of the completion on a given dimension.

4.4.2. Game Mechanics Layer

GameManager:

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 FileManager 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 MouseMapAdapter 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:

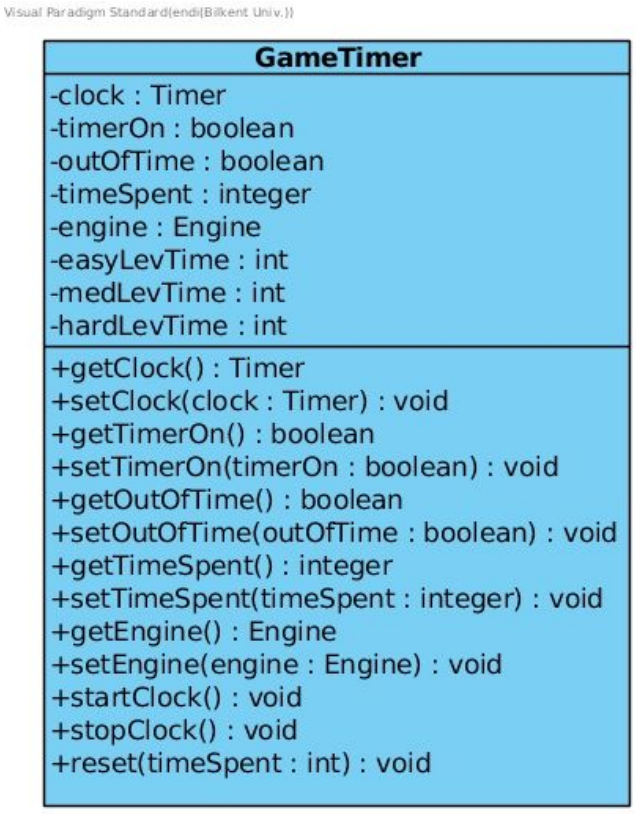
Attributes:

- **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 fileManager 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 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:



Attributes:

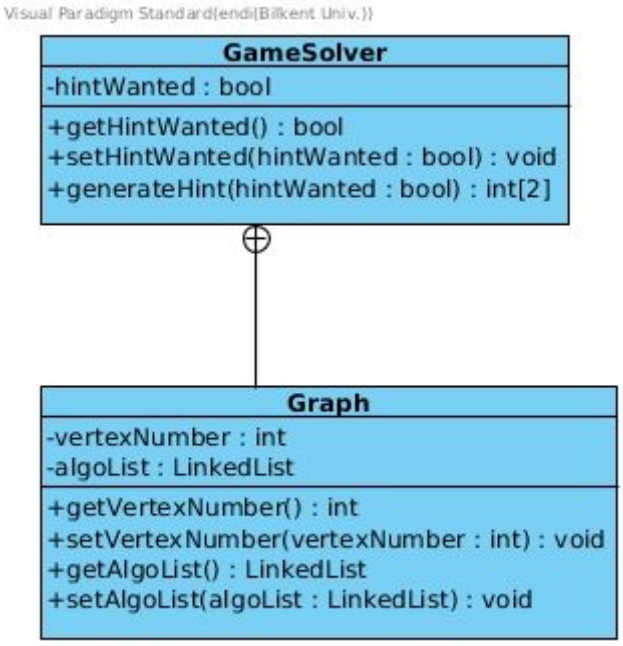
- **private Timer clock:** The java.util class to be used as a timer for the game.
- **private boolean timerOn:** Needed to check if the timer option is enabled or disabled in the game.
- **private boolean outOfTime:** Boolean value to check when the player has run out of time.
- **private int timeSpent:** 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.

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.



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:

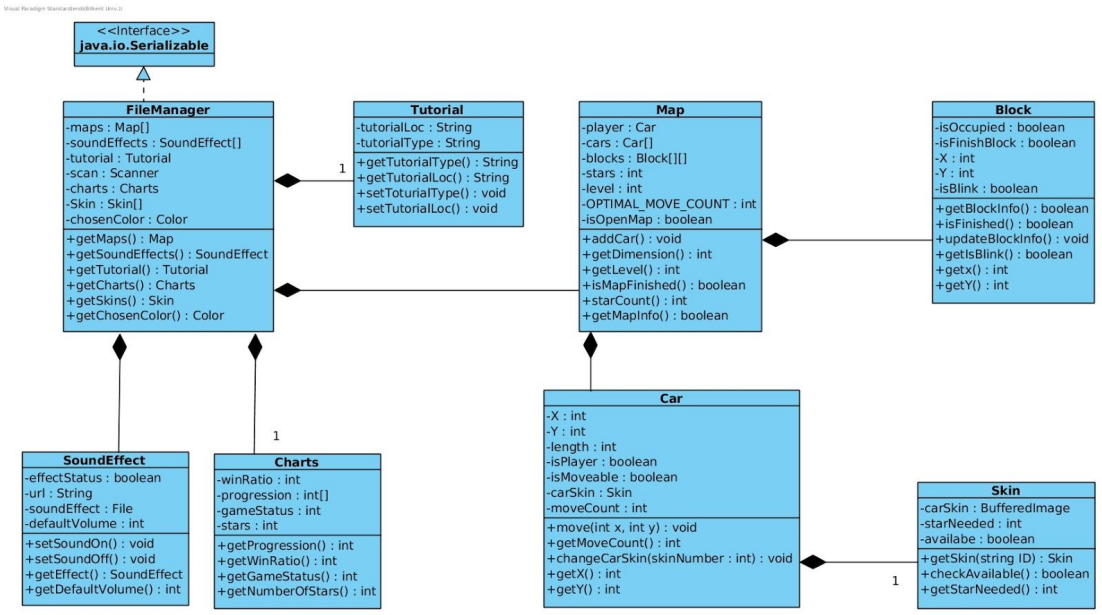
Attributes:

- **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:

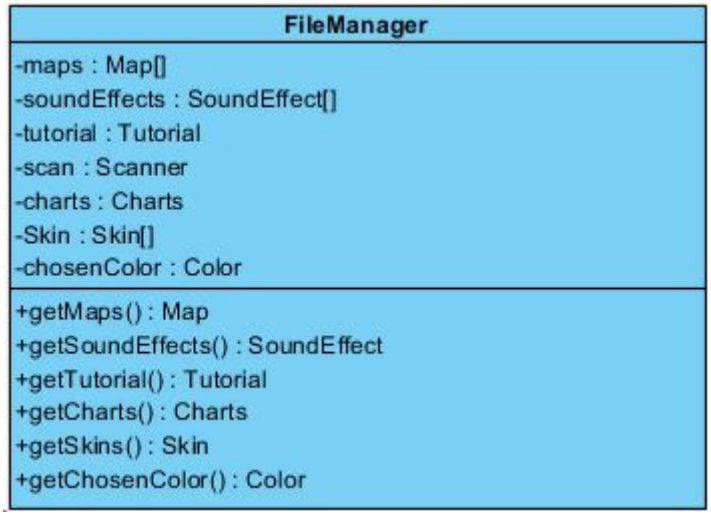
- **public boolean increaseStars():** This will increase the totalStars according to the stars that the player gets from selectedMap.
- **public boolean decreaseStars():** This will decrease the totalStars according to the actions of the player. (The player might get an skin or use his/her stars to get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

4.4.3. Storage Layer



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.

FileManager:



Attributes:

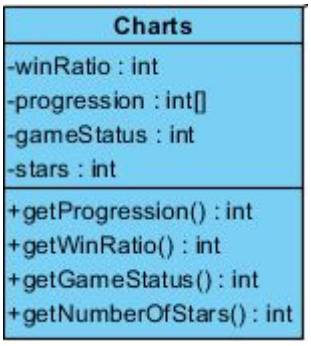
- **private Map maps[]**: File manager will store all the maps as an array.

- **private SoundEffect soundEffects[]**: File manager will store soundEffects as an array.
- **private Tutorial tutorial:** File manager will store the tutorial.
- **private Charts charts:** File manager will store the charts.
- **private Skin skins[]:** File manager will store the skins.
- **private chosenColor:** Game will have a theme color option and chosen theme will be stored in file manager.

Operations:

- **public Map[] maps():** Returns maps.
- **public SoundEffect[] getSoundEffects():** Returns sound effects.
- **public Tutorial getTutorial():** Returns tutorial.
- **public Charts getCharts():** Returns charts.
- **public Skin[] getSkins():** Returns skins.
- **public Color getChosenColor():** Returns chosen color.

Charts:



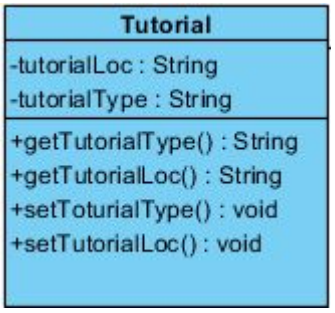
Attributes:

- **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 int getWinRatio():** Returns winRatio.
- **public int[] getProgression():** Returns progression.
- **public int getGameStatus():** Returns gameStatus.
- **public int getNumberOfStars()**: Returns stars.

Tutorial:



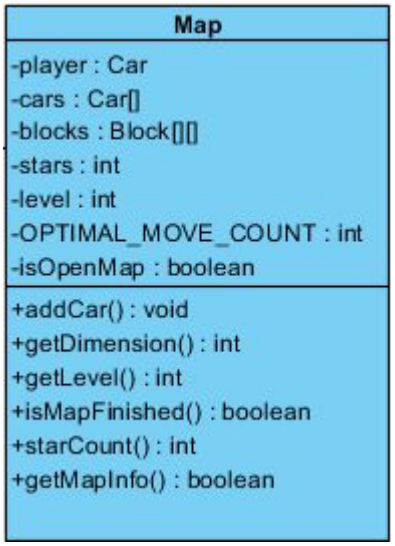
Attributes:

- **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 ():** Sets the location of the tutorial.
- **public void setTutorialType ():** Sets the type of the tutorial.

Map:



Attributes:

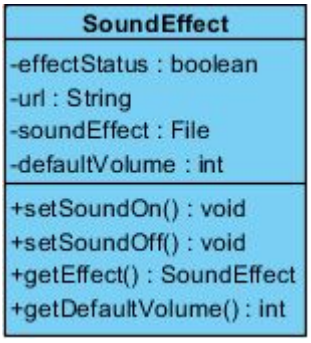
- **private Car player:** Car that tries to leave the park.
- **private Car cars[]:** Moveable cars that block player.
- **private Block blocks[][]:** Two dimensional blocks that create the map.
- **private int stars:** Number of stars earned from that map according to moveCount of the player and optimalMoveCount.
- **private int level:** Level of the map.
- **private final int OPTIMAL_MOVE_COUNT[]:** 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.

Operations:

- **public void addCar():** This will add cars to the map.
- **public int getDimension()**: Returns the dimension of the map according to number of blocks in the map.
- **public int getLevel():** Returns the level of 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.
- **public int starCount():** Returns earned stars.

- **Public boolean getMapInfo():** Returns whether the map is open or not.

SoundEffect:



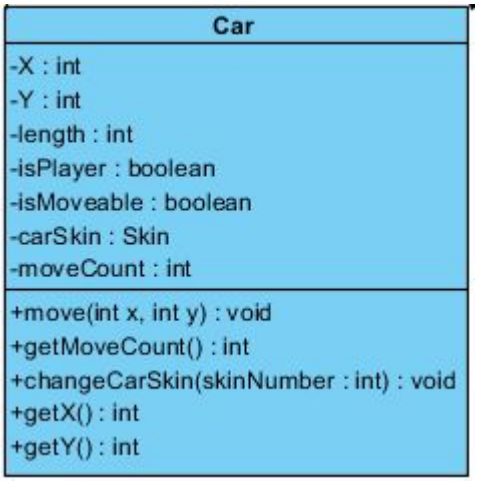
Attributes:

- **private boolean effectStatus:** Is sound effect on or off.
- **private string url:** Url of the sound effect.
- **private File soundEffect**: Actual sound effect file.
- **Private int defaultVolume:** Default volume of the sound effect.

Operations:

- **public void setSoundOn():** Unmute the sound effect.
- **public void setSoundOff():** Mute the sound effect.
- **public SoundEffect getEffect(int effectNumber):** Returns the sound effect.
- **Public int getDefaultVolume():** Returns the default volume.

Car:



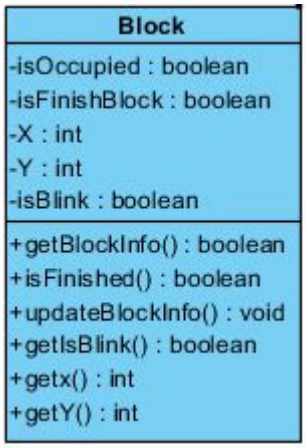
Attributes:

- **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 player or obstacle.
- **private boolean isMoveable:** Car could be moveable 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.

Operations:

- **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 int getX():** Returns the X dimension of the car.
- **public int getY():** Returns the Y dimension of the car.

Block:



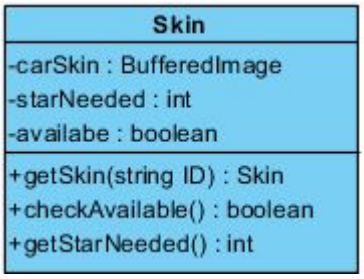
Attributes:

- **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.

Operations:

- **public boolean getBlockInfo()**: Returns isOccupied.
- **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 getIsBlink():** Returns if block is blinking.
- **public int getX():** Returns the X dimension of the block.
- **public int getY():** Returns the Y dimension of the block.

Skin:



Attributes:

- **private BufferedImage 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.

Operations:

- **public Skin getSkin(string ID)**: Returns the desired skin.
- **public boolean checkAvailable()**: Returns if the skin is open.
- **public int getStarNeeded()**: Returns the star needed for opening the skin.

5. *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.