CS319 Object-Oriented Software Engineering
Design Report Iteration-2

RUSH HOUR
GROUP 3.B

Yusuf Dalva
Ahmet Avcı
Melih Ünsal
Musab Gelişgen
Kuluhan Binici

1. Introduction

In this part of the design report, overall design choices are going to be elaborated under the parts of purpose of the system, design goals, end user criteria, maintenance criteria and trade offs between portability and performance.

1.1 Purpose of the System

Rush hour is a sliding block puzzle invented in 1970s[1]. In the board , there are some cars representing the cars in the traffic and the main purpose of this game is taking the target car out by sliding both the target car and the other cars on the board. As an innovative additional feature, hint option is implemented for single player mode. With hint feature, system is always able to know the best next move in the game and user is able to use this feature. However, whenever user uses hint option, he/she lose one star. In addition, there is a multiplayer mode where 2 players trying to taking their cars out first in a board. Finally, we have implemented artificial intelligence bot which players can play against in the multiplayer mode. There are three different difficulty level of bot as easy, normal and hard.

1.2 Design Goals

The game is a kind of mind developing game so in the game we have aimed let the user(s) make a brain exercise and entertain them while they are doing so. To do that, we focus on making the main parts of the game (playing the game) as clear as possible and also the details (hints, ai) which are important for differentiating our game from competitors while these details doesn't directly affect the system. In addition we have used object-oriented design techniques (inheritance, composition) to extend the game easily. The below parts explain the design goal details such as end user criteria, maintenance criteria, performance criteria and trade-offs.

1.2.1 End User Criteria

Usability: From user's point of view, learning a game shouldn't be too difficult, everything should be clear to a user. To achieve this, we didn't let user use any keyboard buttons but we provide user make every possible action by mouse. First, user selects the game mode then play the game by using only mouse.

Performance: For a good game experience, we try to write code as efficient as possible otherwise there would be delays and low fps in the game. To avoid this, we have tried to use the most suitable data structures all over the game learned in CS201 and CS202 (such as graph algorithms while finding the shortest path for hint). In addition, the game does not have so many graphics that force the game to perform poorly.

[1] "Rush Hour®." Thinkfun, www.thinkfun.com/products/rush-hour/.

1.2.2 Maintenance Criteria

Extensibility: The game is designed so that it allows us to add new features in to the game (e.g. the way we added the hint was just adding another class without changing the whole structure which fits the S.O.L.I.D understanding). If change or modification is necessary, these can be handled easily with the design decisions of the game in the future.

Reusability: We have used LibGDX in the game to reuse the functionalities of the engine. We used inheritance to avoid repeating ourselves and came up with separate classes with separate functionalities that allowed us to use some portions of the code over and over again. (e.g. creating map with a given map layout)

Portability: We have chosen Java as a programming language and LibGDX as a library to implement the game because Java provides Java Virtual Machine (JVM) to run in all desktops. LibGDX is also a cross platform library so by only a few changes, an android version of the game could be implemented easily.

1.2.3 Trade-Offs Portability – Performance

In this project, we will use Java as programming language which is compatible with many operating systems and LibGDX as a library which could run in almost all the operating systems by changing only a few lines of code. Thanks to this programming language and library, we built a portable game. However, portability feature of Java makes it slower than those languages which are converted to executables. The extra phases a Java program go through comes with an ignorable run time penalty.

2. Software Architecture

In this part, we are going to explain how we designed the system of the game. To make the game extensible and reusable easily, we constructed system by smaller subsystems.

2.1 Subsystem Decomposition

In this part, we will explain subsystems briefly. We are going to give detailed information of subsystems in the third chapter. Throughout the project, some parts of the subsystem will need to be changed; therefore, we do not combine subsystems rigorously. Otherwise, all subsystems may be affected because of a change occurred in one subsystem. Additionally, such a design with low coupling is useful to improve the game in the future. Briefly, main concerns are modifiability and extensibility. Our system consists of three subsystems: GameLogic Layer, Data Layer and UI Layer. UI Layer is responsible for providing user interface and having interactions with the user. Game Logic Layer is responsible for all decisions and calculations needed for game mechanics. Data Layer will be responsible for storing data about star scores for each completed map.

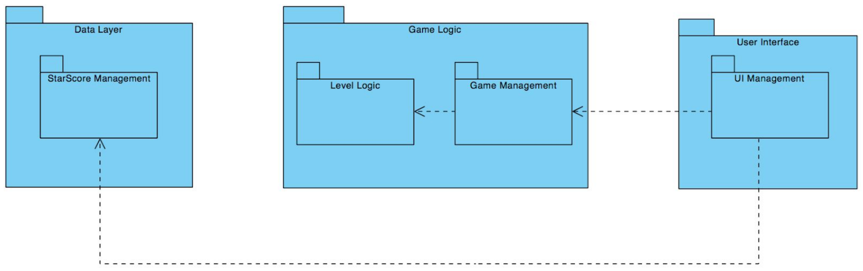


Figure 1. UML representation of system decomposition

2.2 Hardware/Software Mapping

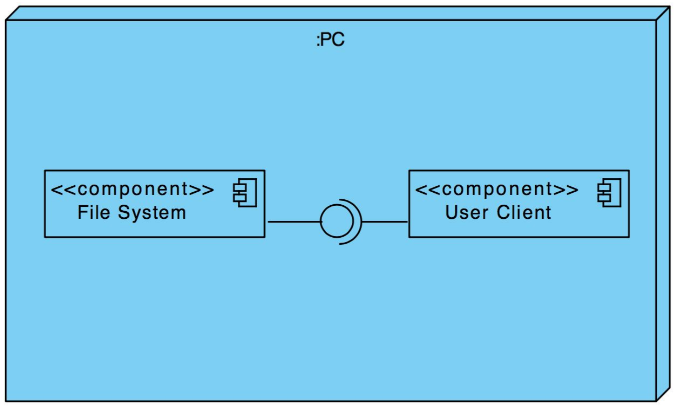


Figure 2. Deployment diagram of our system

The game is going to be implemented in Java. All user interface and graphics components will be constructed with LibGDX game engine which is responsible for all of the graphical issues. Most computers can run this game because it does not contain any 3D rendering and intense calculations. Our game requires a few memory because operations such as serializing the star scores does not require a huge memory. To give inputs, a simple mouse will be enough. Thanks to these simple requirements, many if not all computers can run the game in many operating systems such as Linux , Windows and Mac.

2.3 Persistent Data Management

In the game, star score of each level need to be stored after each game. In order to achieve this, we have used file system by serializing StarScore objects into the file system.

The reasons why we have preferred file system over database are we have raw data which is not complicated, there is no need to share data with multiple platforms and finally we are saving all the users from establishing their own databases unnecessarily (Otherwise, every player who wants to play the game has to setup a database).

2.4 Access Control and Security

Our game does not require any login system yet, because all data is stored locally and do not require any private information from user. Thus, our game does not have any security or access control concerns because there is no authentication for players.

2.5 Boundary Conditions

Execution Rush Hour will be launched by double clicking .jar file of the game. The game does need only Java Runtime Environment and LibGDX library files.

Termination:
Scenario 1: Player clicks the exit button in the main menu.
Scenario 2: The game can be terminated by clicking X icon on the window.

Failure:
Scenario 1: If there is an error with database system, the user will not be able to save the progress he has made or even lose his progress.
Scenario 2: If electricity or computer problem occurs, last situation of the game is not saved. Thus the user will lose all progress.

3. Subsystem Services

In this part, design choices about subsystem structure are going to be elaborated.

3.1 Game Logic

This section contains the general model classes of the game. In this subsystem, GameManagement package is in charge of keeping the track of current game conditions. The BaseGame class in it has an attribute of Skin type which handles the relationship of objects to the file system assets (such as knowing which image to get from file system for drawing a specific car on the screen). Other subpackage is LevelLogic which controls the layout of the maps as well as the Car instances for each map. Moving Car instances and drawing wild cards are also handled by this package with appropriate methods.

3.2 User Interface

User Interface package has the UI components. Initial UI component when the game opens is MainMenuPanel. UIManager is in charge of opening other panels as the user performs actions. This package communicates with Game Management subpackage and StarScore Management subpackage. BasePanel class is the ancestor of all other panes in the package which are displayed on the screen according to the user actions.

The panels in the game are: high score panel, help panel, music panel, theme selector panel, car selector panel, single/multiplayer levels panels, etc. When necessary (by user requests), panels will be created and loaded onto the screen.

3.3 Data Layer

Data Layer manages the star scores through serialization. It communicates with UI package to be updated when necessary (when a single player game ends).

4. Low Level Design

In this section, low level design choices are going to be elaborated.

4.1 Object Design Trade-offs

The object design required to apply some of the fundamental object-oriented design processes in order to perform the implementation process in the most efficient way possible. In the game the objects are designed in an easily manageable and reusable (e.g. level system) way to deal with overall complexity. Data that is needed to be stored in the game is being kept as a local storage (e.g. StarScore) which gives an advantage of easy access. This design may cause ignorable time loss, yet it still counts as a space-time tradeoff.

4.1.1 Reusability Of The Objects

In order to manage the overall complexity of the application, the program is divided into three subsystems respectively. The three subsystems were involving:

- User Interface provided to the user
- Game logic which enables the game to proceed by keeping game state
- Data Layer which is responsible from the data that is stored locally

In each of these, the design of these systems were done in an hierarchical way that the classes can be managed in an easily and meaningful way.

4.1.1.1 Use of Inheritance

As a fundamental concept of object-oriented design philosophy, inheritance stands at a crucial spot with respect to the concepts that the application promises to provide the user. The game that is subjected in this project includes different levels for different user modes. With respect to this concept introduced, there are SinglePlayerLevels and MultiPlayerLevels classes inheriting the Levels class. Using this logic helps reducing complexity during level selection and game initialization. As a result, the logical construction of the Levels objects benefit greatly from the existence of inheritance. In the subject of user interface provided to the user, the visible representation of the game uses panels in order to display different parts of the game. In that matter the use of Graphical User Interface classes were compulsory and the implementation of different panels can be done easily by forming an extension logic with the BasePanel class we created as the parent of all panel classes. Each panel included in the game consists of different qualities and responses to user actions. With the existence of inheritance, the predefined qualifications can be redefined and used easily. As explained for the panels included in the game, other GUI components included in the game also benefit from the use of extension logic in the management of components contained in the user interface provided to the user.

4.1.1.2 Interfaces and Serialization of Object Data

As another concept to provide reusability, the presence of interfaces is important in the program. Even though there are no specific interfaces that are going to be implemented by the developers of the project, the Serializable interface provided by Java makes great contribution to the data storage system of the program. In order to save the instances generated during the execution of the project saving them as object instances are not possible. This interface provides a way to store these objects that contributes to the storage system of the software. As shown in the class diagram of Data Layer subsystem includes object instances that are needed to be stored locally. Which is the star score data in this case. This local storage is enabled by the Serializable interface that brings up the easiness of not defining methods to serialize the objects repeatedly in the implementation.

4.1.1.3 Polymorphism

As defined in the section regarding the use of inheritance, polymorphism is used as an object-oriented concept. The panels constructed are derivations of default panel class provided by Java which result with completely different panel instances, that is clearly is another use of polymorphism in the solution domain.

4.1.2 Space-Time Tradeoffs

For the storage system of the software, the data is stored locally and uses the Serializable interface of Java to write the data. The data written is not stored in a compressed way which results with a tradeoff in terms of space used by the program. Using the Serializable interface provided by Java is also another constraint that is resulting with a tradeoff. As the interface provided results with lower performance there is a loss of efficiency in the program. However, in general vith the use of both non-compressed data storage and the use of Serializable interface results with easier implementation and less complex logic. With respect to all of these aspects, the overall process of data storage can be considered as space-time tradeoff.

4.1.3 Encapsulation

For security purposes of the system and to ensure that the players are not able to change anything about the game logic or any subsystem included in the game, encapsulation is another concept that is introduced in the project. The process of encapsulation results with extra implementation efforts (use of extra resources), but the security is an important tradeoff for the sustainability of the system. With the existence of encapsulation, the designed classes can only share the information they store in the way that is implemented in the Game Logic. In summary, the information that the user shall not access are all labeled as private and they can just be shared with the predefined ways in solution domain.

4.2 Final Object Design

The detailed view of the object model is given below for a more clarified view. The object model given below contains three subsystems of the project that are: Data Layer, User Interface and Game Logic.

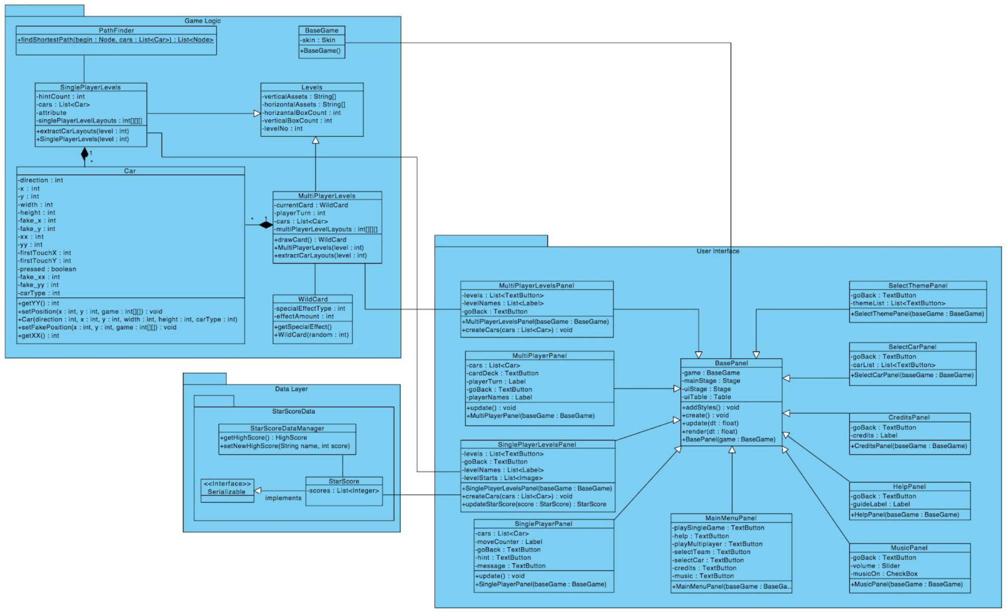


Figure 3. UML class diagram of our system

4.3 Packages

4.3.1 User Interface Package

The User Interface Package is mainly for providing visual content required for the game. This graphical content is composed of panels that has different kinds which is related to the functionalities it provides to the user. These panels which are specific to the functionalities that it provides to the user are used for the different displays that the game provides. The occurrence of these panels are provided by the User Interface subsystem.

For this subsystem the BasePanel class may be considered as the most crucial class as it inherits all of the other panels that are provided by this subsystem.

Specifically, BasePanel class is the parent class of all of the classes that are provided by the subsystem. As a structure it represents the panels used in the game in general. This class enables the creation of the screens that are provided to the user, updating the visual content according to the user actions and defining styles to specific panels respectively.

The detailed view of the User Interface package is given below:

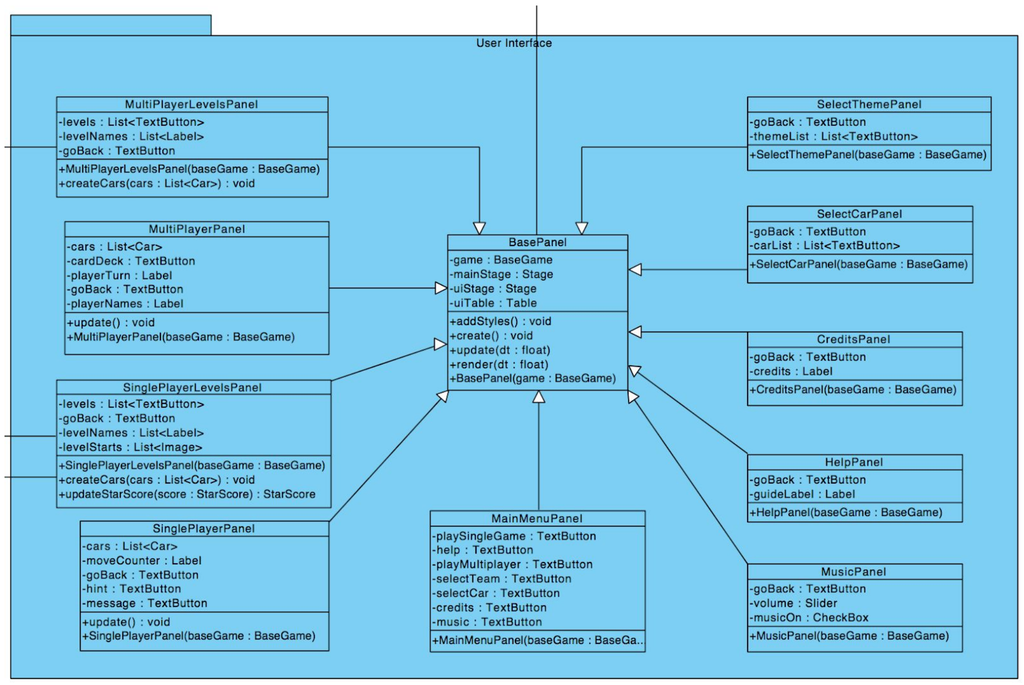


Figure 4. UML representation of the User Interface Package

4.3.1.1 BasePanel Class

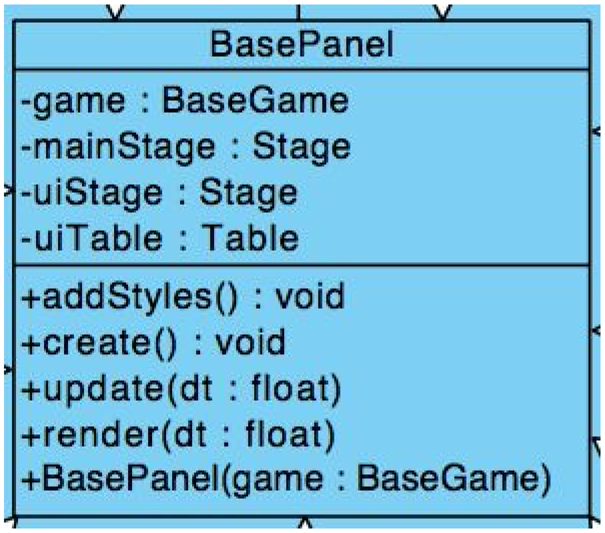


Figure 5. BasePanel class

BasePanel class is responsible for providing child classes to take an input from mouse and also allows page changes when the appropriate buttons are pressed. The class achieves these by implementing Screen and InputProcessor interfaces. It also generates all the skins for user interface elements.

Attributes:

protected BaseGame game: That is the object that all the classes responsible for changing a screen, have to take it as a parameter in their constructors.

protected static Stage mainStage: It contains all the non user interface elements in the game.

protected static Stage uiStage: It contains all the user interface elements and also a Table.

protected Table uiTable: It contains all the user interface elements and contained by uiStage.

Methods:

public void addStyles(): It creates all the skins for both ui and non-ui elements.

public boolean keyDown(int keycode): It is called when keyboard is pressed.

public void render(float dt): renders the game by 60 fps

public void update(float dt) it updates all the changes in the game and called by render method.

public boolean isPaused(): Returns true if the game is paused.

public void setPaused(boolean b): Determine whether the game is paused or not.

public void togglePaused(): It pauses the game if the game is not paused and it continues the game if the game is paused.

public boolean keyUp(int keycode) : It is called when the keyboard is released.

public boolean keyTyped(char character): It is called when something is typed by keyboard.

public boolean touchDown(int screenX, int screenY, int pointer, int button) : It is called when mouse is clicked in desktop game or the screen is touched in the mobile version.

public boolean touchUp(int screenX, int screenY, int pointer, int button) :It is called when mouse is released in desktop version or when we remove our finger from screen in the mobile version.

public boolean touchDragged(int screenX, int screenY, int pointer) : It is called when we drag the mouse in desktop version or when we drag our finger in the android version.

public boolean mouseMoved(int screenX, int screenY) : It is called when the mouse is moved.

public boolean scrolled(int amount) : It is called when the we scroll the mouse.

public void resize(int width, int height): It is called when the window size is changed.

4.3.1.2 MultiPlayerLevelsPanel Class

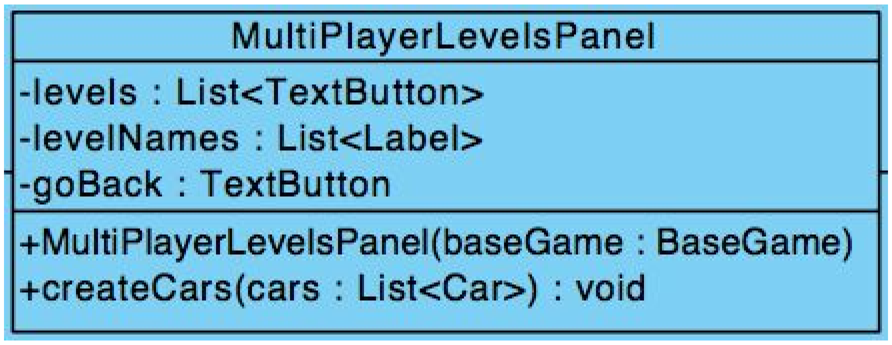

MultiPlayerLevelsPanel
-levels : List<TextButton>
-levelNames : List<Label>
-goBack : TextButton
+MultiPlayerLevelsPanel(baseGame : BaseGame)
+createCars(cars : List<Car>) : void

Figure 6. MultiPlayerLevelsPanel class

The MultiPlayerLevelsPanel class is the structure which is going to be responsible for providing the user an interface that shows the levels that can be played by the user in multiplayer mode of the game. The interface provides information about the playability of the levels due to the current progress of the player.

Attributes:

- private List<TextButton> levels: For the representation of the level names as text buttons, this attribute provides sufficient information. This is a list of Text Buttons which contains one button for each level which the user can click on it open a specific level if it is unlocked.
- private List<Label> levelNames: As well as the text buttons, for each level there are labels which contains information about the name of the level. With these labels, the name of each level is going to be represented to the player.
- Private TextButton goBack: In order to enable the user to exit from this screen a go back button is provided as a text button. By pressing this button the player may exit from the MultiPlayerLevels panel

Methods:

- protected MultiPlayerLevelsPanel(BaseGame baseGame): This is the constructor for MultiPlayerLevelsPanel. It takes a BaseGame instance which is member of the Game logic package. The BaseGame class contains the skin that is going to be used in the game and by using that skin the panel can be initialized
- protected void createCars(List<Car> cars): By this method, the arrangement of the cars are settled and the level that is going to be constructed according to the user choice is going to be arranged. The game instance present in the BasePanel is going to be modified according to this method.

4.3.1.3 MultiPlayerPanel Class

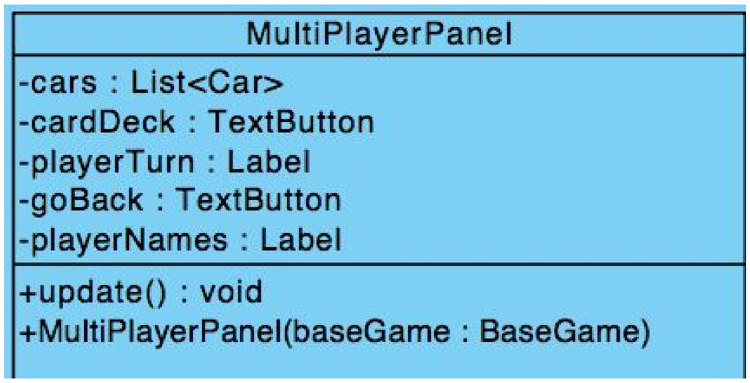


Figure 7. MultiPlayerPanel class

This is the class for visualizing the MultiPlayer mode levels to the user. By using this screen the user can make moves according to the visual representation provided.

Attributes:

- private List<Car> cars: The cars that are going to be included in this level is going to be holded in this property. This list contains the arrangements for each car present in the game
- private TextButton cardDeck: As the multiplayer mode consists of wildcards as a feature, by using this button the user can request to select a wildcard and the card is going to be dealt to the user as a result of this request
- private Label playerTurn: The multiplayer mode consists of two sides who can play against each other. With this label the player who is going to be playing is going to be shown
- private TextButton goBack: In order to exit from this panel, the goBack TextButton is used.
- private Label playerNames: The names of the sides playing the multiplayer mode is shown using this label

Methods:

- protected MultiPlayerPanel(BaseGame baseGame): This is the constructor for the MultiPlayerPanel class. It takes a BaseGame which includes the sufficient information about the level that is going to be costructed
- protected void update(): By using this method, after each move attempt of the player, the visual shown in the panel is updated. As a result the panel is responsive to user moves.

4.3.1.4 SinglePlayerLevelsPanel Class

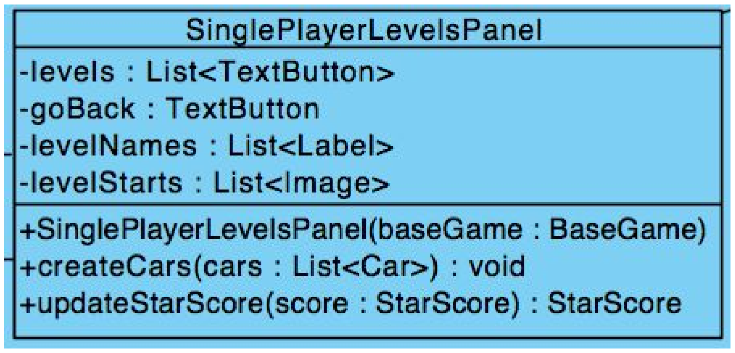

SinglePlayerLevelsPanel
-levels : List<TextButton>
-goBack : TextButton
-levelNames : List<Label>
-levelStarts : List<Image>
+SinglePlayerLevelsPanel(baseGame : BaseGame)
+createCars(cars : List<Car>) : void
+updateStarScore(score : StarScore) : StarScore

Figure 8. SinglePlayerLevelsPanel class

This is the class which shows the levels contained in the Single Player mode. The user can access to these levels by using this panel.

Attributes:

- private List<TextButton> levels: The levels can be accessed using the TextButton's representing the levels. By using the TextButton's stored in this property the user can make a selection for the level that the player wants to play
- private TextButton goBack: In order to exit from this panel in the game, then user can use the goBack TextButton

- private List<Label> levelNames: The names of the levels are represented in a textual way in the panel. By the instances of this property, the names of the levels can be provided to the player.
- private List<Image> levelStarts: There is a visual representation for each level that is implemented in the game. The user can distinguish them by the Images that represent them. The ımages of each level are provided by this property.

Methods:

- protected SinglePlayerLevelsPanel(BaseGame baseGame): This is the constructor for SingelPlayerLevelsPanel. The information needed for construction is provided by the BaseGame class.
- protected void createCars(List<Car> cars): The cars that are going to be created in each level are hold in this list. The list contains the configurations for each car in each level.

4.3.1.5 SingleplayerPanel class

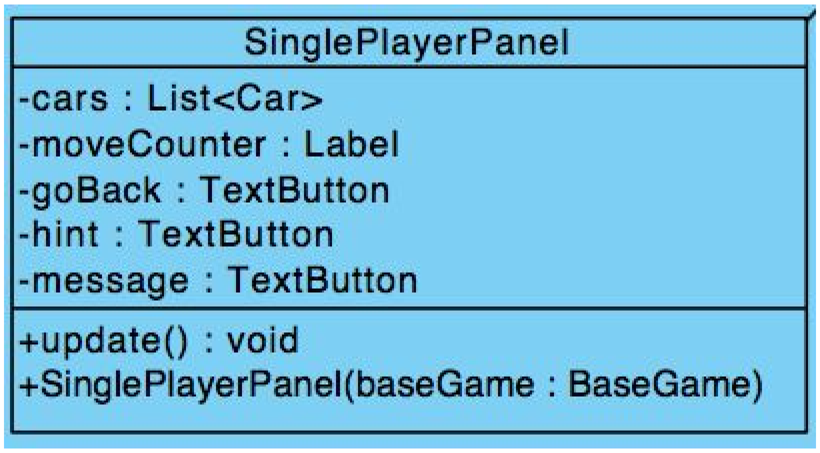


Figure 9. SinglePlayerPanel class

The SinglePlayerPanel provides the visual screen to the user which enables the user to see the panel that contains the single player game and enables the player to make moves accordingly (successfully playing the game).

Attributes:

- private List<Car> cars: the cars that are going to be contained in this level are hold in this attribute, the information about the configuration of the car in the level is also provided by this attribute
- private Label moveCounter: As the moves that are being made in the mode determines the stars collected in the game, the moveCount is being holded and the user should be able to see that information. By this Label the user can see the moves that he/she made starting from the initialization of the panel.

- private TextButton goBack: To exit from the game, the user uses this goBack TextButton which allows exiting from the single player game
- private TextButton hint: The game provides hints to the user to pass the level. By using this button the user requests an hint and a move is done by the system which is according to finish the game successfully.
- private TextButton message:

Methods:
- protected SinglePlayerPanel(BaseGame baseGame): This is the constructor for the SinglePlayerPanel class. It takes a BaseGame instance as it provides the sufficient information to construct the visual for the level.
- protected void update: According to the moves that are being made by the user, the game is going to be updated. By the use of this method, visual representation of the level is being updated.

4.3.1.6 MainMenuPanel class

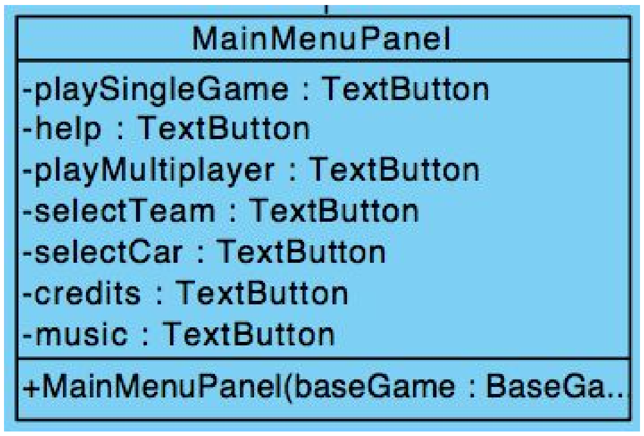


Figure 10. MainMenuPanel class

The main menu of the game which enables the user to navigate between game modes and settings of the game is provided by this visual provided to the user.

Attributes:
- private TextButton playSingleGame: The user can navigate through the single player game using this TextButton instance
- private TextButton help: The user can navigate through the help screen using this button, which is the page that gives information about how to play the game
- private TextButton playMultiplayer: Using this TextButton the user can navigate through the Multiplayer game and play it.
- private TextButton selectTeam: For the multiplayer game, the user can select a team using this TextButton

- private TextButton selectCar: The escape car that is going to be used in the game is selectable. By using this TextButton, the user can make this selection
- private TextButton credits: The credits given in the game is illustrated in another panel. Using this instance, that screen is accessible
- private TextButton music: The settings for the music of the game can be adjustable with another panel. By using this instance that panel is accessible

Methods:

- protected MainMenuPanel(BaseGame: baseGame): This is the constructor for this class.the baseGame instance provides sufficient information about the game and MainMenuPanel class is constructed accordingly.

4.3.1.7 SelectThemePanel class

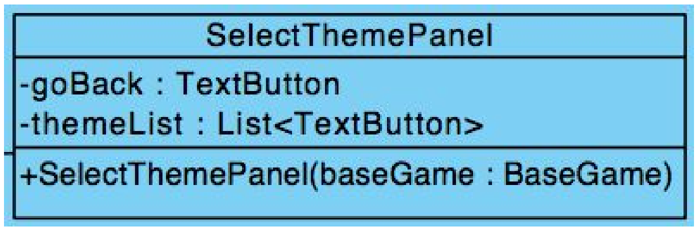

SelectThemePanel
-goBack : TextButton
-themeList : List<TextButton>
+SelectThemePanel(baseGame : BaseGame)

Figure 11. SelectThemePanel class

In the game, the user can decide on the theme that the game is going to be played on. By this panel, the theme of the game can be updated by the user.

Attributes:

- private TextButton goBack: In order to close this panel, the user can use the goBack TextButton which navigates through to the previous panel that was rendered.
- private List<TextButton> themeList: The themes provided in the game are selectable by the TextButtons provided in this List, for each theme there is a TextButton that is registered to this list

Methods:

- protected SelectThemePanel(BaseGame baseGame): This is the constructor for the SelectThemePanel class. It takes a baseGame instance to retrieve information about the themes present in the game and construct the list from the themes. After the construction process, the user can make the selection which affects the game logic directly

4.3.1.8 SelectCarPanel class

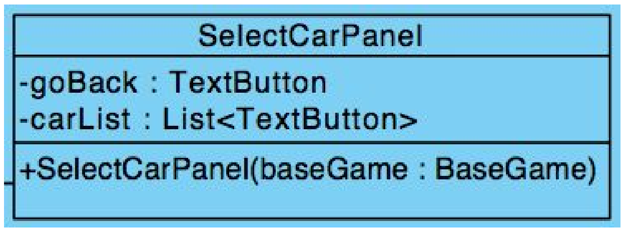

SelectCarPanel
-goBack : TextButton
-carList : List<TextButton>
+SelectCarPanel(baseGame : BaseGame)

Figure 12. SelectCarPanel class

The user can select the car that he/she wants to play as, which going to be the escape car in the game. This panel enables the car selection which is going to be effecting the game logic directly due to the connection with BaseGame instance

Attributes:

- private textButton goBack: The exit functionality from the panel is supplied with this textButton, by pressing this button the user can go back to the panel that is rendered previously
- private List<TextButton> carList: The cars that are selectable are provided as TextButtons which are stored in this list. There is a TextButton registered in the list for all of the possible cars that can be selected

Methods:

- protected SelectCarPanel(BaseGame baseGame): As the constructor of the SelectCarPanel class, it takes a BaseGame instance which provides the connection with the Game Logic package. According to the data retrieved from this instance, the TextButtons for the cars that are selectable are constructed.

4.3.2 Data Layer Package

The Data Layer package is for handling the data storage. This package contains a subpackage which is constructed only for clarity. The levels of single player game have a score system which assigns stars according to the moves performed by the user and according to predetermined constraints, the boundaries of star scores are determined. As the user plays the game there are specific star counts that are collected (player has a highscore in terms of star count for each level). These star counts are stored in the StarScoreData package by convention. The setting operation of high score and getting that high score is handled by StarScoreManager class. The StarScore instance is an integer list which holds and integer list that holds the star counts for each level. This data is stored as local data, which is the reason that it implements the Serializable interface.
The detailed view of Data Layer package is given below:

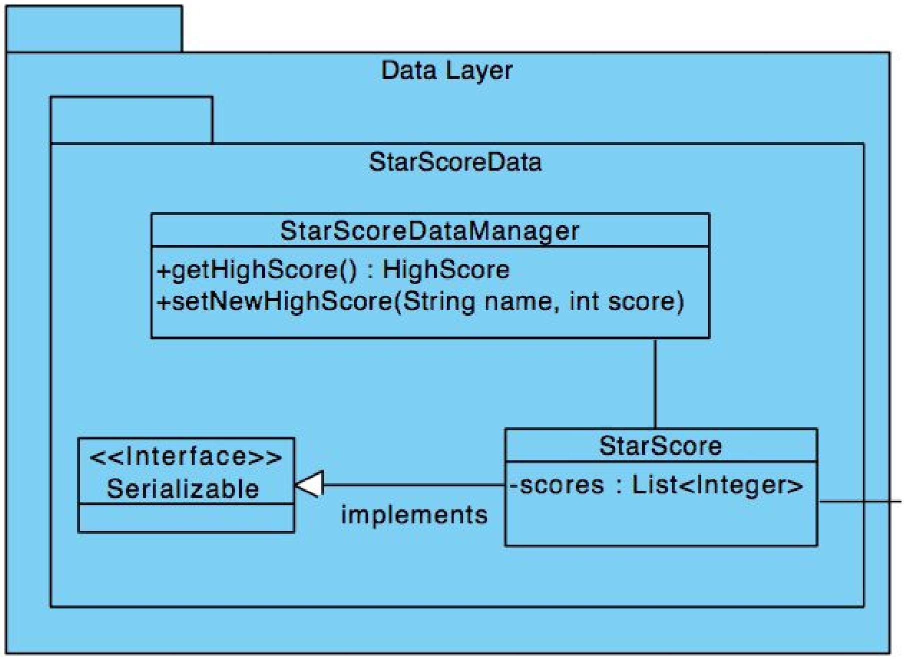


Figure 13. Data Layer subsystem

4.3.2.1 StarScoreDataManager class

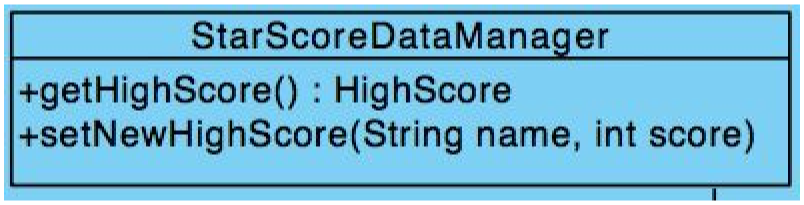

StarScoreDataManager
+getHighScore() : HighScore
+setNewHighScore(String name, int score)

Figure 14. StarScoreDataManager class

As a manager class, StarScoreDataManager class contains only two methods for updating and retrieving data. The explanations of these methods are given below

Methods:

- protected StarScore getStarScore(): This method is for retrieving the star score data(list containing the count of stars collected for each level) for all of the present levels in the single player mode. The method returns a StarScore instance which is the form that the data is stored.
- protected void setNewStarScore(String name, int score): For updating the star score(number of maximum stars collected in a level changes) data this method is used for updating the StarScore object stored. The name attribute taken as a string input gives information about the level and the score data is the new high score for the specified level. With respect to these attributes taken as input, the High Score data that is stored locally is updated.

4.3.2.2 StarScore class

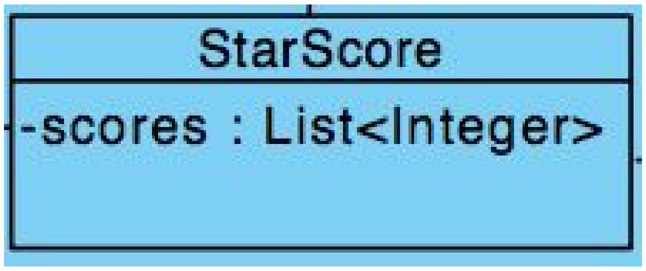


Figure 15. StarScore class

This class is the form of storage of data containing the amount of stars collected for each level in the local storage system. In order to save the data stored in this class this class uses the Serializable interface. The description of data storage is given in the attribute information provided below.

Attributes:

- private List<Integer> scores: The data regarding to the stars collected each level is stored as an integer list object. The scores instance includes the star score data of the single player mode levels.

4.3.3 Game Logic Package

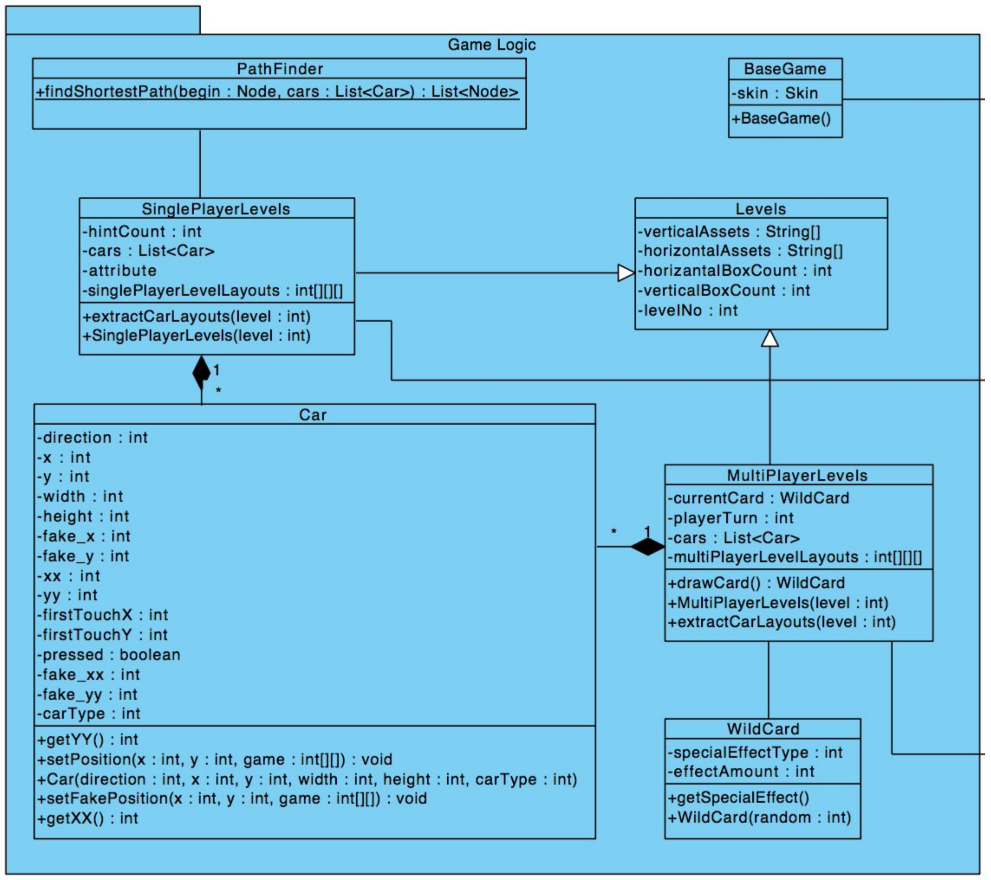


Figure 16. Game Logic subsystem

This is the package for the implementation of the logic of the game. This subsystem is composed of 7 classes which all have crucial importance. These classes are PathFinder, BaseGame, SinglePlayerLevels, Car, Levels, MultiPlayerLevels and WildCard

The BaseGame class stores the skin that is going to be used in the panels which establishes connection with the User Interface package. SinglePlayerLevels and MultiPlayerLevels classes store the data regarding the construction of the levels (configuration of the cars and information about the map of the levels). As they both contain information about levels of the game, they inherit the Levels class which is the parent class of them.

For the Car class, the properties of the cars are stored in this class. This class also controls the success of the moves of the car instances and makes the desired moves logically. Also all of the characteristic information about the cars are stored in this class.

As the final component of the Game Logic package, the WildCard class contains specialized moves that is going to restrict the moves of the player in multiplayer mode. All of the information regarding to the restriction is being stored in this class.

4.3.4.1 BaseGame class

All the classes that responsible for creating screen needs to take an instance of this class as an input parameter in their constructor so that they can reach all the skins created in BasePanel class. It also extends Game class in LibGDX library to provide the authorized objects set the screen when needed.

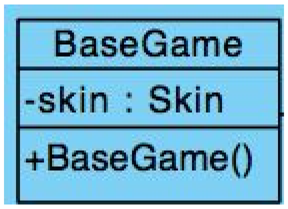


Figure 17. BaseGame class

Attributes:

Skin skin: It enables objects to get the skins created in BasePanel class when needed.

Methods:

public void dispose(): It is called when the program terminates.

4.3.4.2 Levels Class

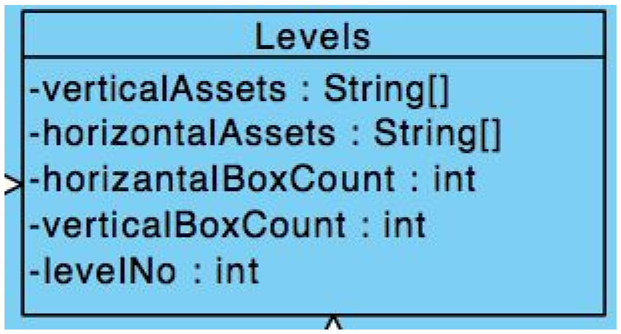

Levels
-verticalAssets : String[]
-horizontalAssets : String[]
-horizantalBoxCount : int
-verticalBoxCount : int
-levelNo : int

Figure 18. Levels class

As the general structure for the levels, this class provides the functionalities to the SinglePlayerLevels and MultiPlayerLevels classes as their parent class

Attributes:

- private String[] verticalAssets: The assets in the game are mapped according to their file names. In each level the assets that have a vertical position are stored in this instances, where they are stored with their file names.
- private String[] horizontalAssets: Just like the vertical assets stored in the game, the horizontal assets are mapped using their file names. These file names representing the assets are stored in this instance
- private int hoizontaBoxCount: With this integer value, the horizontal dimensions of the level is determined
- private int verticalBoxCount: With this integer value, the vertical dimensions of the level is determined
- private int levelNo: As the levels are distinguished with their level numbers, this integer holding the level number distinguşishes the levels for each game mode.

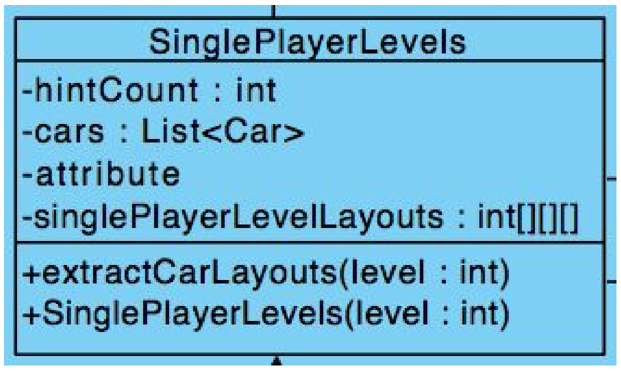

SinglePlayerLevels
-hintCount : int
-cars : List<Car>
-attribute
-singlePlayerLevelLayouts : int[][][]
+extractCarLayouts(level : int)
+SinglePlayerLevels(level : int)

Figure 19. SinglePlayerLevels class

The SingelPlayerLevels class has the information about the setup of all of the levels present in Single Player game mode. As it is as well a level, it inherits the Levels class.

Attributes:

- private int hintCount: This instance holds the count of the hints used in the game
- private List<Car> cars: The list of the cars contained in a specific level is being hold with this instance
- private int[][][] singlePlayerLayouts: The layout of all the levels included in the game is holded in this three dimensional array. (level information, Layout information, car information)

Methods:

- protected void extarctCarLayouts(int level): By using this method the cars attribute can contain the list of the cars specified in singlePlayerLevellayouts atrşbute. The level input of the method specifies the level which is going to be extacted.
- protected SinglePlayerLevels(int level): With the desired levels, the SinglePlayerLevels class is being constructed. (containing information about the cars at a given level)

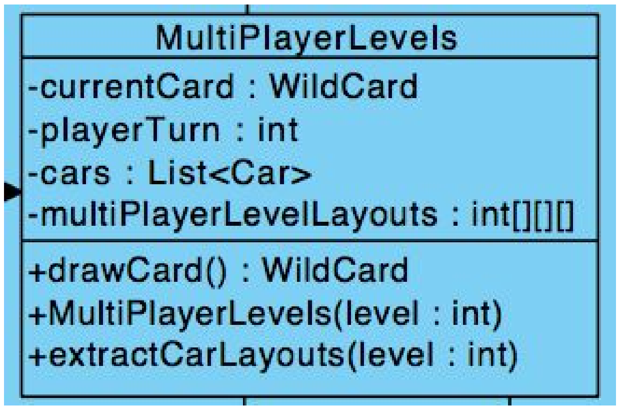

MultiPlayerLevels
-currentCard : WildCard
-playerTurn : int
-cars : List<Car>
-multiPlayerLevelLayouts : int[][][]
+drawCard() : WildCard
+MultiPlayerLevels(level : int)
+extractCarLayouts(level : int)

Figure 20. MultiPlayerLevels class

This class contains specific information about each level in Multiplayer mode. As it contains information about the levels, Levels class is the parent class of this class.

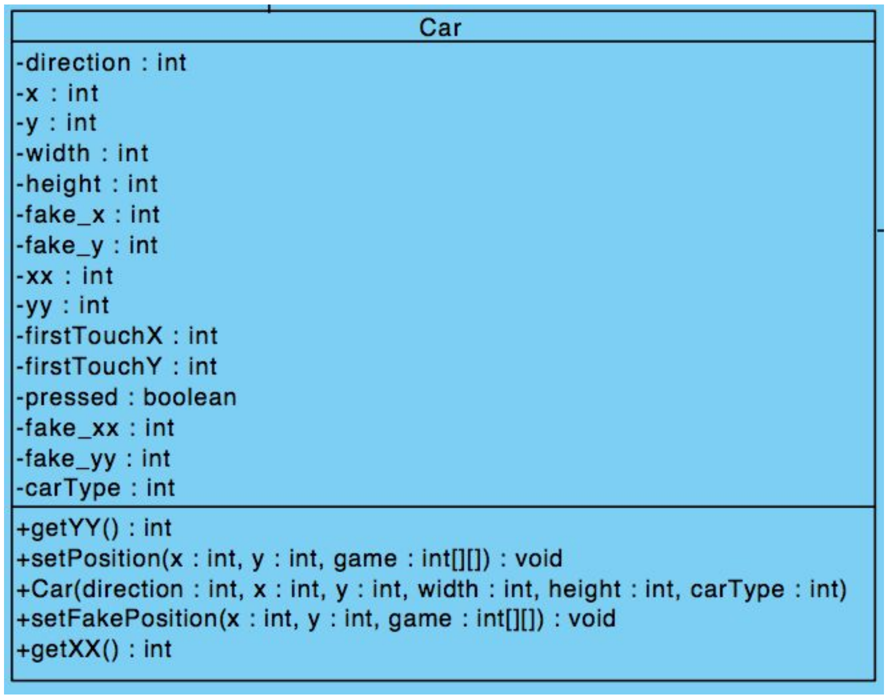
Attributes:

- private List<Car> cars: The cars present in the map are stored with this attribute
- private int playerTurn: This integer value specifies, which player has the right to play at the current turn
- private int [][][] multiPlayerLevelLayouts: Information related to all of the levels present in the Multiplayer mode is stored in this variable just like the principle used in SinglePlayerLevels class

Methods:

- protected WildCard drawcard(): During the multiplayer mode, the user can draw a wildcard using this method. Returns a wildcard instance
- protected void extractCarLayouts(int level): The cars at a specific levels are extracted into the cars attribute using this method.
- private MultiPlayerLevels(int level): With the desired level, the MultiPlayerLevels class is being constructed which has the relevant information about that specific level (Car information)

4.3.4.3 Car class



```
Car
-direction : int
-x : int
-y : int
-width : int
-height : int
-fake_x : int
-fake_y : int
-xx : int
-yy : int
-firstTouchX : int
-firstTouchY : int
-pressed : boolean
-fake_xx : int
-fake_yy : int
-carType : int
+getYY() : int
+setPosition(x : int, y : int, game : int[][]) : void
+Car(direction : int, x : int, y : int, width : int, height : int, carType : int)
+setFakePosition(x : int, y : int, game : int[][]) : void
+getXX() : int
```

Figure 21. Car class

Attributes:

- private int x1: The x coordinate of one of two points stored for the car is stored as an integer
- private int x2: The x coordinate of the other of the two points for the car is stored as an integer
- private int y1: The y coordinate for the first of two point for the car is stored as an integer.
- private int y2: The y coordinate for the other of the two points for the car is stored as an integer.
- private int xx: The x coordinate that the car is wanted to put. It equals to x when it is in the needed boundary.
- private int yy: The y coordinate that the car is wanted to put. It equals to y when it is in the needed boundary.
- private int firstTouchX: It is the starting x coordinate when it is started to be dragged.
- private int firstTouchY: It is the starting y coordinate when it is started to be dragged.
- private fake_x : It is another x coordinate of the car when pathfinding algorithm is running.
- private fake_y : It is another y coordinate of the car when pathfinding algorithm is running.
- private boolean pressed: It becomes true when the car is pressed by mouse and becomes false when mouse is released.

- Private carType: It is 1, if the car is escape car and 0 if not.

Methods:

- protected int[] getLocation(): This method returns the coordiantes of the two points stored for a particular car.
- protected void setLocation(int[] coordinates): This method enables changing the location of the car with the coordinate information provided as an input
- protected boolean contains(int x, int y): This method checks whether the boundaries of the car contains the given point (x,y). Returns true if the car contains the specified point.
- public void setPosition(int x,int y,int[][] game): It changes the location of the car if possible.
- public void setPosition(int x,int y,int[][] game): It is used to change fake_x and fake_y for pathfinding if possible.

4.3.4.4 WildCard class

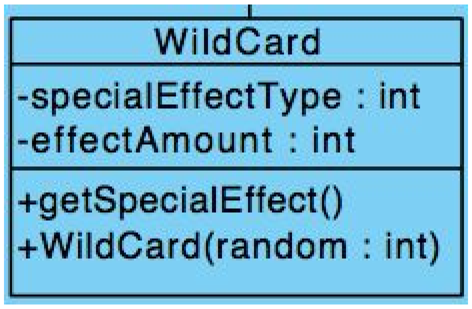


Figure 22. WildCard class

Attributes:

- private int specialEffectType: With this integer the type of the special effect provided by the WildCard is encoded.
- Private int effectAmount: This integer the number of units can be moved with respect to the determined special effect.

Methods:

- protected int getSpecialEffect(): This method enables to access to the special effect specified in the wildcard. The special effect is returned as a string.
- protected Wild Card(int random): With the specified random integer, the special effect for the WildCard is being determined and accordingly, with a random effectAmount, a wildCard instance is being constructed with this constructor.

4.3.4.5 PathFinder class

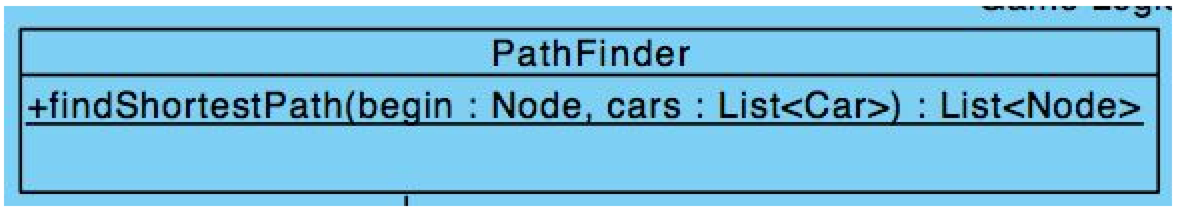

PathFinder
+findShortestPath(begin : Node, cars : List<Car>) : List<Node>

Figure 22. PathFinder class

- public static List<Node> findShortestPath(begin: Node, cars: List<Car>): This static method implements A* algorithm (similar to dijkstra's algorithm) to find the shortest path to the end by using the given car list. The path is returned as a linked list.

4.4 Class Interfaces

As stated in section 4.1, the class uses the serializable interface for enabling local data storage. In this motivation, StarScore class is going to be implementing Serializable interface. Due to the requirement of the method signatures defined in the Serializable interface, the StarScore class is going to be implemented with

- private void writeObject(java.io.ObjectOutputStream out): Enables to store the object output to the local data specified
- private void readObject(java.io.ObjectInputStream in): Enables to read object data from the local data stored.
- private void readObjectNoData() throws ObjectStreamException: Enables to check whether reading from the local data storage is possible or not. If not possible throws an exception.

5. Improvement Summary

In general, the object-class model of the system had to be changed notably because of the game engine we are using. As we progressed in our code, we changed several things in our design because of technical reasons and then revisited our diagrams to apply those changes. We simplified the game logic subsystem of our game. We removed the enumeration classes that we previously included for operations like selecting skin or selecting theme. We merged escape car and obstacle car classes into Car class for simplicity. We used inheritance in representing game levels such as single player levels and multi player levels (now they both extend Levels class). We added the hint feature which came along with the PathFinder class that allows us to find the shortest possible solution to solve the maze from the current location of the player's escape car. We removed the MapData and MapDataManager classes since we decided that they are not necessary. There are not much important changes in our UI subsystem. To sum up, the changes we made, are done in order to provide simplicity, to add new features or to fix some technical problems and we can say that we have improved our game's performance.

6. Glossary & References

1- What Every Computer Scientist Should Know About Floating-Point Arithmetic, 6 Oct. 2018, docs.oracle.com/javase/7/docs/api/java/io/Serializable.html.

2- Rush Hour®." Thinkfun, www.thinkfun.com/products/rush-hour/.

3- "Rush Hour® Shift." Thinkfun, www.thinkfun.com/products/rush-hour-shift/.