CS319 Object-Oriented Software Engineering Design Report Iteration-1

RUSH HOUR

GROUP 3.B

Yusuf Dalva
Ahmet Avcı
Melih Ünsal
Musab Gelişgen
Kuluhan Binici

1. Introduction

1.1 Purpose of the System

Rush hour is a sliding block puzzle invented in 1970s. 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. In addition, we have added some extra features such as a immovable objects like trees or stones in the board and also there is a multiplayer mode where 2 players trying to taking their cars out first in a board. There is also a new feature that players may segment the board by a limited number ways obeying some rules in the multiplayer mode to ease their jobs or prevent their opponents from taking his car out first. By these new features and good graphics,, we aim to whet users' appetite for this game. When implementing the game, we don't forget the guiding principles which are the object-oriented design techniques.

1.2 Design Goals

The game is a kind of mind developing game so in the game we have aimed let to user(s) make a mind exercise and entertain them while they are doing it. To do that, we focus on making the main parts of the game as good as possible and also the details which is important for differentiating our game from others while doesn't directly affect the system. In addition we have used object-oriented design techniques 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. In addition, in the game, when user wants to pause the game or look at the instructions in the help menu, an overlay menu appear on the screen so that user doesn't need to pass window for each time.

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 cs 201 and cs 202. In addition, the game does not have so many graphics that force the game to perform poorly but there is a few but we have handled by the techniques in the previous courses.

1.2.2 Maintenance Criteria

Extensibility: The game is designed so that it allows us to add new features in the feature. If there is a time remained we aim to add some extra features to multiplayer mode and also more cars and large maps as well. We can do it easily thanks to the extensibility of our design.

Reusability: We have used LibGDX in the game to increase the users' desire to the game in terms of graphical qualities and so many classes we have implemented by using the library, can be used for other games by not changing as well easily.

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) which doesn't give any problem in many operating systems. LibGDX is also a cross platform library so by only a few changes, we can implement an android version of the game easily.

1.2.3 Trade-Offs Portability – Performance

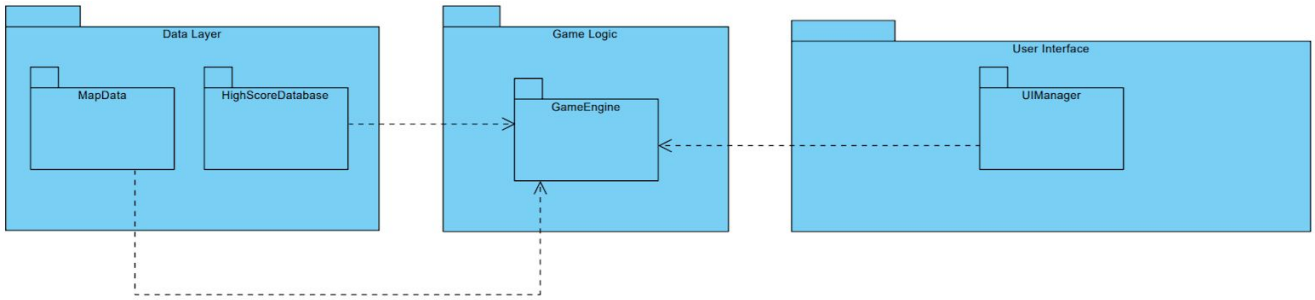
In this project, we will use Java as programming language which compatible with many operating systems and LibGDX as a library which also runs in almost all the operating systems by changing only a few line of code. Thanks to this programming language and library, we build a portable game . However, it also means that Java will go through several phases before it converts the Java code to machine code. This phases may occur some performance penalties.

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. In our project, we will use Model View Controller (MVC) design which is suitable for this kind of game project.

2.1 Subsystem Decomposition

In this part, we will explain subsystems briefly. We are going to give detailed information in the third chapter. Throughout the project, some part 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 is useful to develop the game in the future. Briefly, main concerns are modifiability and extensibility. Our system consists of three subsystems:Game Logic Layer and Data Layer and GUI Layer. GUI 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 high scores and maps.



2.2 Hardware/Software Mapping

Our game is going to be implemented in Java programming language. All user interface and graphics components will be constructed by using Java libraries such as LibGDX library which is responsible for most of the graphical issues. Most computers can run this game because it does not contain any 3D rendering and intense calculations. Our game requires only a few memory because operations such as saving high scores, map and game does not require a huge memory. To give inputs, basic mouse will be enough. Thanks to these simple requirements, many computers can run the game in many operating systems such as Linux , Windows and mac.

2.3 Persistent Data Management

In the game, high scores, result of multiplayer games, created map and current position of a map are permanent data that needs to be stored after each game. We will use local database to store all of them since our game doesn't have any network connection yet. It means, user can only see high scores which obtained his/her own computer and the results played in the multiplayer mode in the same computer. By using file system, data will be written into csv. Files.

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. We may extend it later so that the game needs to have a login system but at the moment it doesn't need. Therefore, the project does not require any access control or security.

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

This part contains explanations about subsystems of Rush Hour.

3.1 Game Logic

This section contains the general model classes of the game. In this subsystem, GameEngine is in charge of keeping the track of current game conditions. GameEngine has attributes such as the current level the player is playing, which gamemode is being played, player's turn (modified in Multiplayer only) highscores and game stop/pause conditions. It also has functions to control this attributes as well as the game flow (e.g. drawing wild cards, moving platform, moving cars, etc.) Map and WildCard classes composes GameEngine. Map has a theme and many cars. Car has two types which could be either EscapeCar or ObstacleCar.

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 the game logic through GameFrame. GameFrame is the frame of Rush Hour and displays currently active panel according to the UIManager's commands.

The panels in the game are: high score panel, help panel, music panel, theme selector panel, car selector panel, single/multiplayer map selection panel, ingame panels, etc. When necessary (by user requests), panels will be created and loaded onto the screen. HighScorePanel interacts with Data Layer to get high scores and displays them in the screen.

3.3 Data Layer

Data Layer manages the high scores through serialization. It also manages map data by reading files and sending the map data to the GameEngine when necessary.

4. Low Level Design

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 way to deal with overall complexity. Data that is needed to be stored in the game is being kept as a local storage which gives an advantage of easy access. However, for easy access the data is kept in an uncompressed way which is also the cause of loss in execution time (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
- 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 several types of cars and aiming to escape from the maps where the constructed cars are placed. With respect to this concept introduced, the cars stated are implemented using inheritance taking a basic car as a superclass. Using this logic helps reducing complexity of the types of cars. As a result, the logical construction of the car 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 frame class that the programming language provides. 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 high 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. Since the logical existence of car objects require the presence of superclasses, the subclasses are formed as a derivation of the superclass and form different object instances which is a clear use of polymorphism. As another use of polymorphism, 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

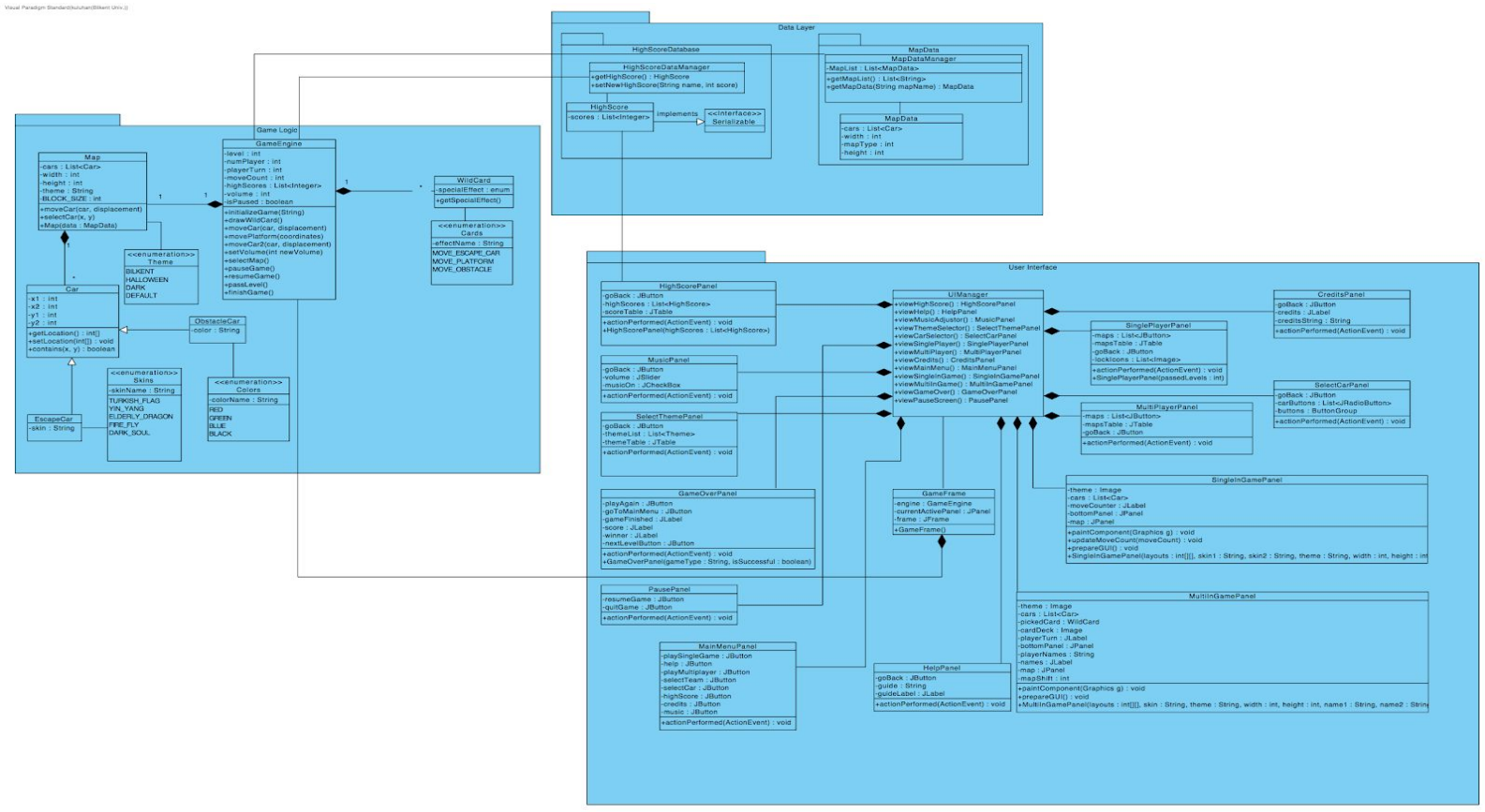
For the storage system of the software, the data is stored locally and uses the Serializable interface of Java to write the data. The data written is not stored in a compressed way which results with a tradeoff in terms of space used by the program. Using the Serializable interface provided by Java is 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 the general view 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



4.3 Packages

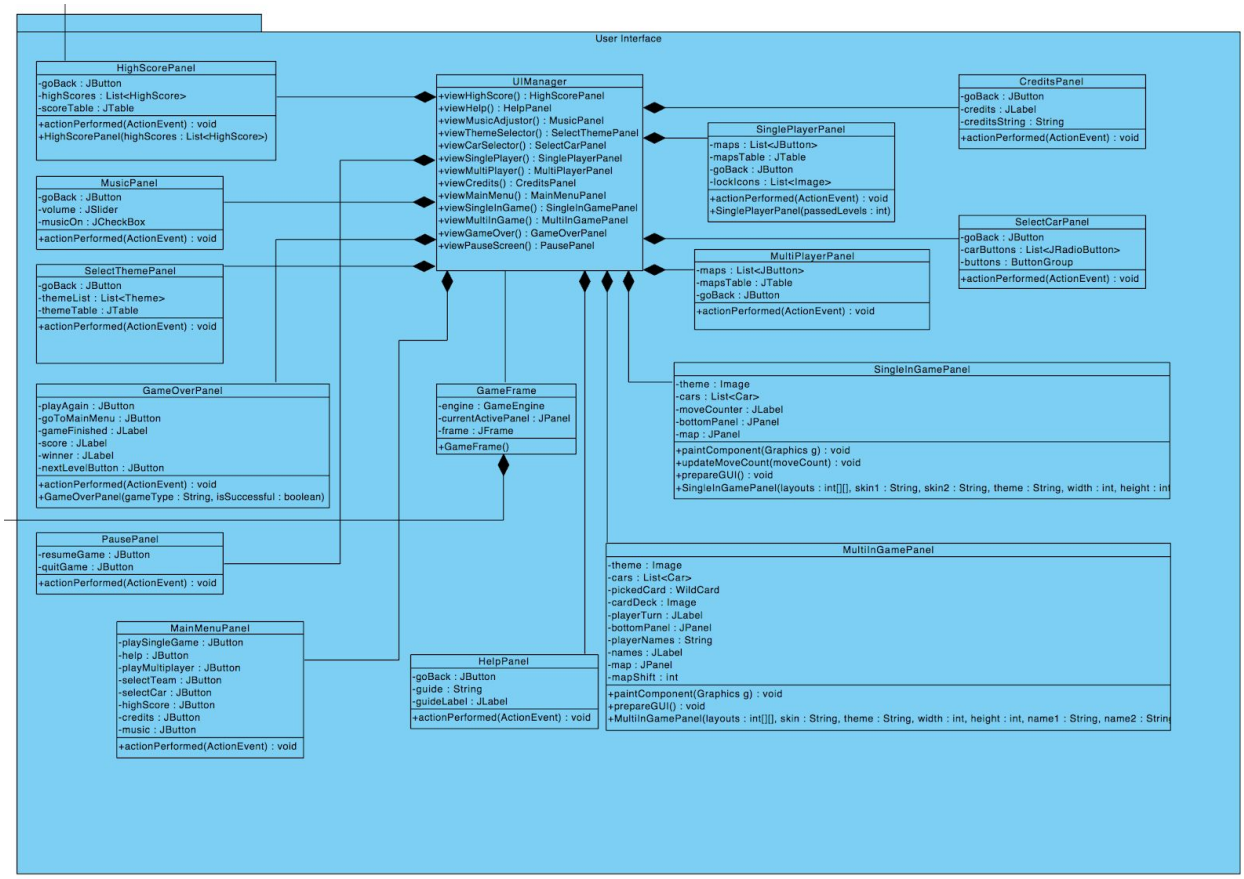
4.3.1 User Interface Package

This package is for providing the subsystem the graphical content required. This graphical content includes the main frame that the game is being displayed and the panels which are content specific. These content specific panels are used for different displays which are managed from User Interface subsystem.

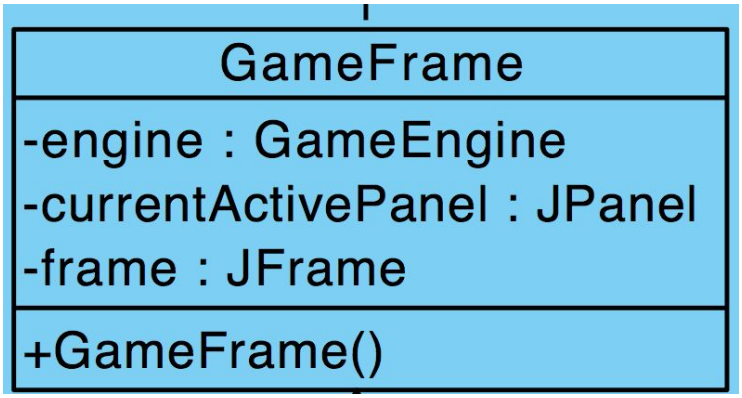
For this subsystem there are two crucial classes. These classes are UIManager and GameFrame classes.

UIManager is the service class of the User Interface subsystem which provides the GameFrame class the panel required according to the action that the user wants to perform. Class has functionality to display each of the panels with the defined methods. The clear description of these functionalities are given in this section.

GameFrame is the class that contains the frame that the user displays the game with the panels generated with the help of UIManager class. The class also interacts with the Game Logic subsystem with an instance of Game Engine. The clear description of the solution domain provided is explained in this section.



4.3.1.1 GameFrame Class



Attributes:

- private JFrame frame: This is the JFrame instance that is going to display all of the visual contents included in the game
- private JPanel currentActivePanel: Since the panels that the user is going to interact with is going to be determined by the UIManager class, there is going to be a panel that is going to be active at the Frame. This instance represents the panel that the user is interacting with that is included in the frame instance.
- private GameEngine engine: The GameFrame interacts with the Game Logic subsystem by using this instnace. This engine instance holds every detail that are specified with the logic of the game. This class takes user inputs and performs necessary actions respectively.

Constructor:

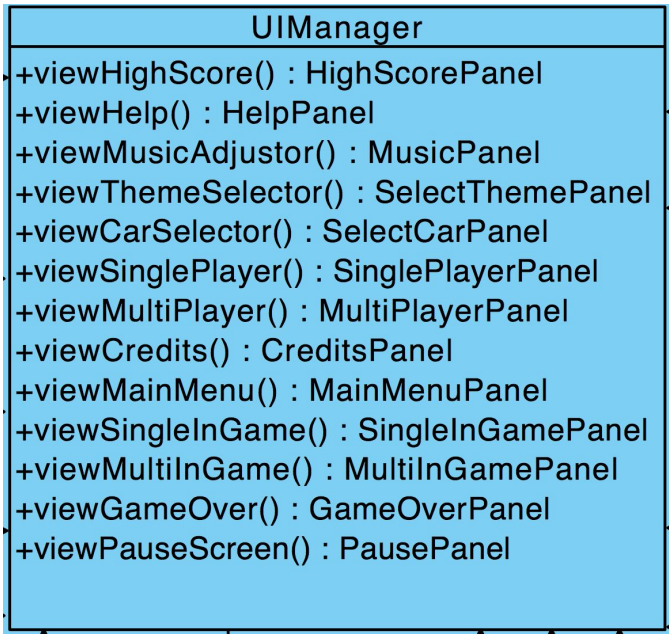
- public GameFrame(): The constructor is called each time that a GameFrame object is initialized.

1 - After the call to construction frame, engine and currentActivePanel instances are initialized

2 - With the initialization the user is going to be directed to the Main Menu panel. In order to accomplish that the Main Menu Panel is assigned as the currentActivePanel as the user is going to interact with it initially.

GameFrame class does not include any methods.

4.3.1.2 UIManager Class



UIManager class only contains methods.

Methods:

- protected HighScorePanel viewHighScore(): With the use of this method, the UIManager can create and return a HighScorePanel. The HighScorePanel has the functionality of showing the high score for the current level specified in the GameEngine instance.
- protected HelpPanel viewHelp(): This method generates and returns a specific panel named HelpPanel. This HelpPanel instance is used for displaying the guide for the game to the user.
- protected MusicPanel viewMusicAdjustor(): This is the method that is designed to generate and return a MusicPanel for the user to interact with. MusicPanel enables the user to adjust the sound options of the program.
- protected SelectThemePanel viewThemeSelector(): By this method the SelectThemePanel can be generated and returned to be used in the GameFrame class. By interacting with SelectThemePanel instance the user can select a theme and apply that theme to the game.
- protected SelectCarPanel viewCarSelector(): This method enables the class to generate and return a SelectCarPanel instance. By using SelectCarPanel instance, the user can make a selection for the escape car that is going to be representing the user in the game.

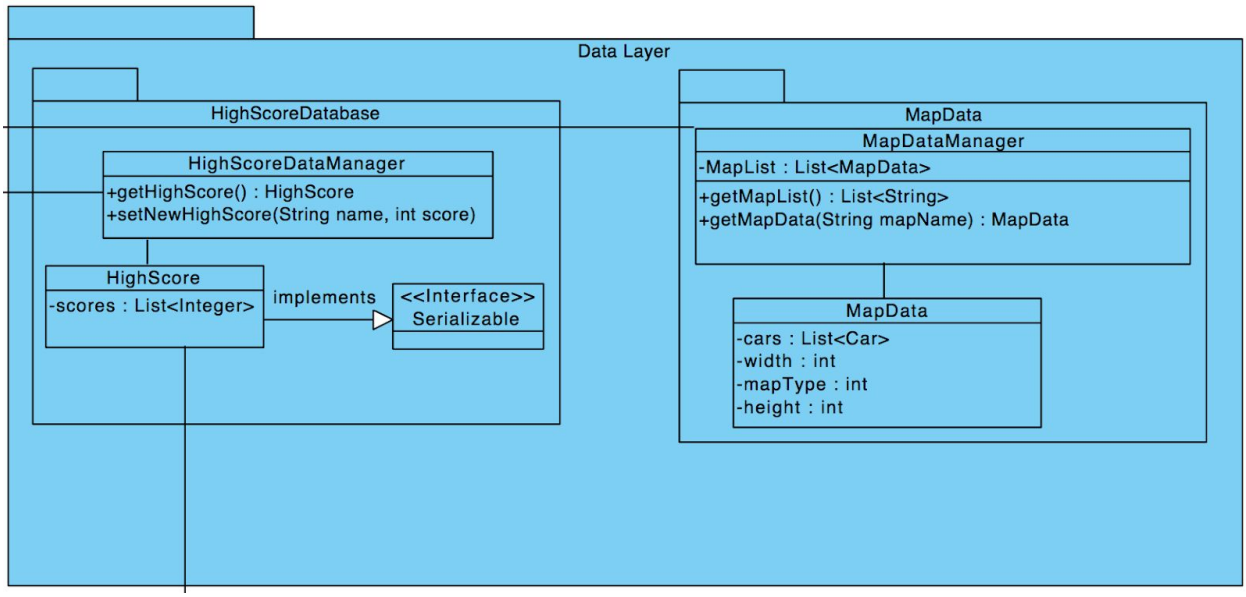
- protected SinglePlayerPanel viewSinglePlayer(): With the use of this method the SinglePlayerPanel is returned. This panel enables the user to make selections for the Single Player Mode. These preferences are taken into consideration when the Single Player game is initialized.
- protected MultiPlayerPanel viewMultiPlayer(): This is the method for generating and returning MultiPlayerPanel instance. This instance shows the options for the Multiplayer Mode of the game and the user can make selections about the game. When the Multiplayer Mode is being initialized the selectins that the user made are taken into consideration.
- protected CreditsPanel viewCredits(): By this method the CreditsPanel can be instantiated and returned. With the use of CreditsPanel instance the information about the developers of the application can be displayed.
- protected MainMenuPanel viewMainMenu(): This method enables the application to generate and return MainMenuPanel instance. By this instance, the main menu of the game is displayed where the user can perform specific actions that are navigated with the use of this panel. The actions that the user can start to perform at this panel are playing Single Player Game, playing Multiplayer Game, selecting a car, displaying the high score in the game, selecting a team, displaying sound options, displaying help options, and displaying the credits screen. The actions performed also involve UIManager class.
- protected SingleInGame viewSingleInGamePanel(): This method enables the creation and returning of the SingleInGamePanel instance. The Single Player Mode of the game is performed with use of this panel. The move count, theme and the drawing of the game is contained in this panel.
- protected MultiInGamePanel viewMultiInGame(): This method enables the creation and returning of the MultiInGamePanel instance. By using this instance the Multiplayer mode of the game can be performed. The panel contains the visualization of the game board, wildcards, the valid turn and all the other details regarding the multiplayer mode of the game.
- protected PausePanel viewPauseScreen(): With this method the user can generate the PausePanel instance and return it. By the use of PausePanel method the user can pause the game, quit the game completely and resume the game from this panel.

4.3.2 Data Layer Package

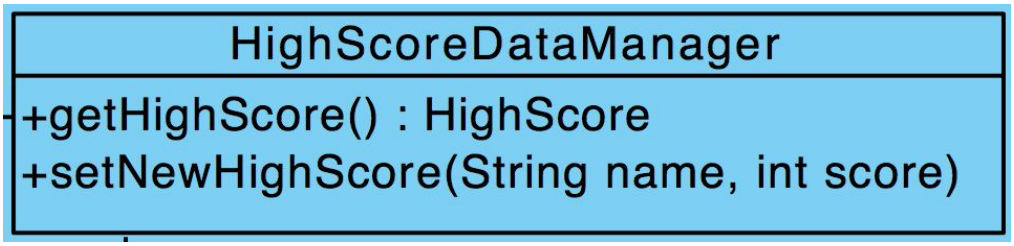
This is the package that the handling of data storage is performed. This package contains two subsystems for data management. These subsystems are HighScoreDatabase and MapData. These systems are present basically for getting updates from the information stored in these systems(Map data and highscore). These packages can mangle all the necessary actions about the management of data, that are save, get and update. For each

system there is a manager class present, which are HighScoreDataManager and MapDataManager respectively.

The detailed view of Data Layer package is given below:



4.3.2.1 HighScoreDataManager class

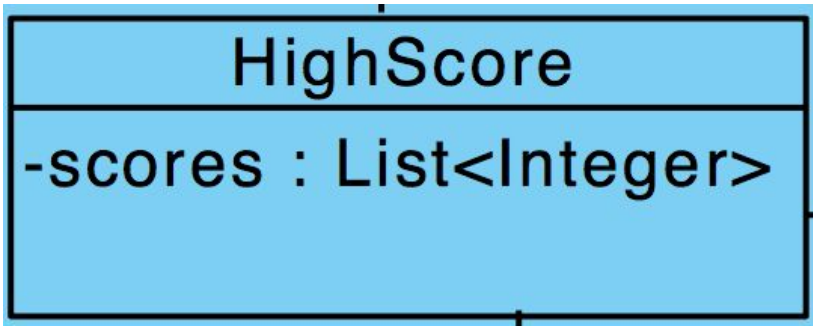


As a manager class, HighScoreDataManager class contains only two methods for updating and retrieving data. The explanations of these methods are given below

Methods:

- protected HighScore getHighScore(): This method is for retrieving the high score data for all of the present levels in the single player mode. The method returns a HighScore instance which is the form that the data is stored.
- protected void setNewHighScore(String name, int score): For updating the high score data this method is used for updating the high score 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 HighScore class

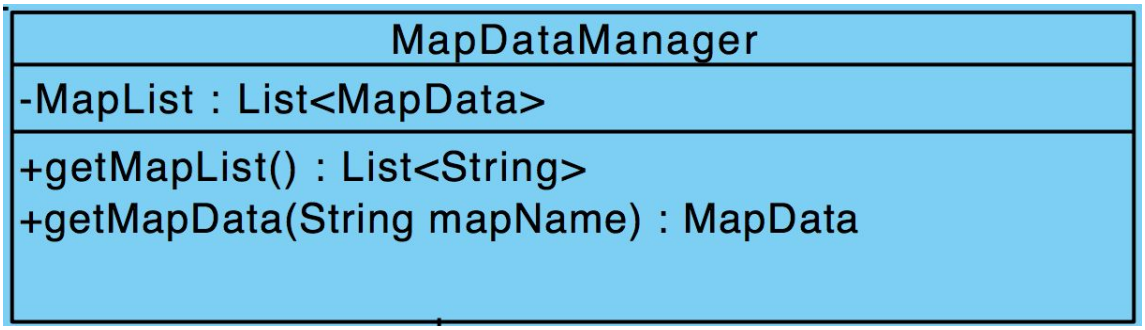


This class is the form of storage of high score data in the local storage system. In order to save the data stored in this class this class uses the Serializable interface. The description of data storage is given in the attribute information provided below.

Attributes:

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

4.3.2.3 MapDataManager class



As another class involved with data management, the class only involves getting and setting methods to access and modify the data stored locally. The contents of the class are given below.

Attributes:

- private List<MapData> MapList: The list of the data regarding to the maps are stored in the MapList list in the form of MapData instances.

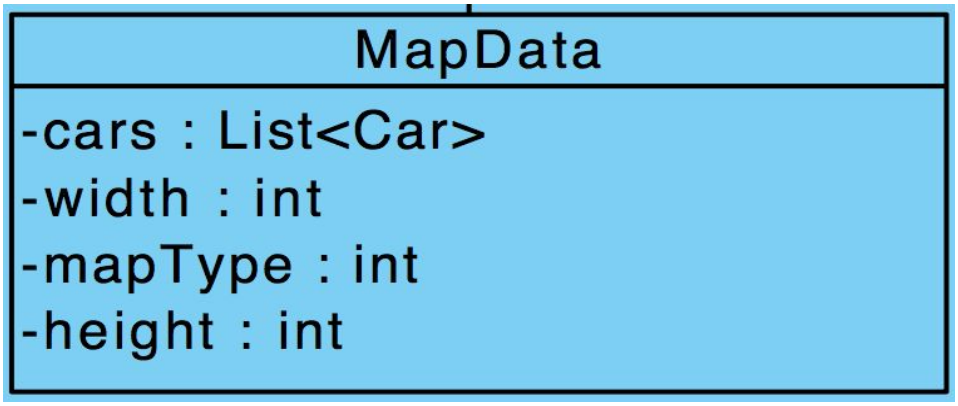
Methods:

- protected List<String> getMapList(): As the data involving the maps included in the game is stored in the form of a MapData list, this method enables the access to this list.

Respectively, getMapList() methods returns the string representation of the MapData instances in a form of a list.

- protected MapData getMapData(String mapName): With a specified map name as a string input, this method allows to get the regarding data about the specified map. Method returns a MapData instance with the desired map name.

4.3.2.4 MapData class

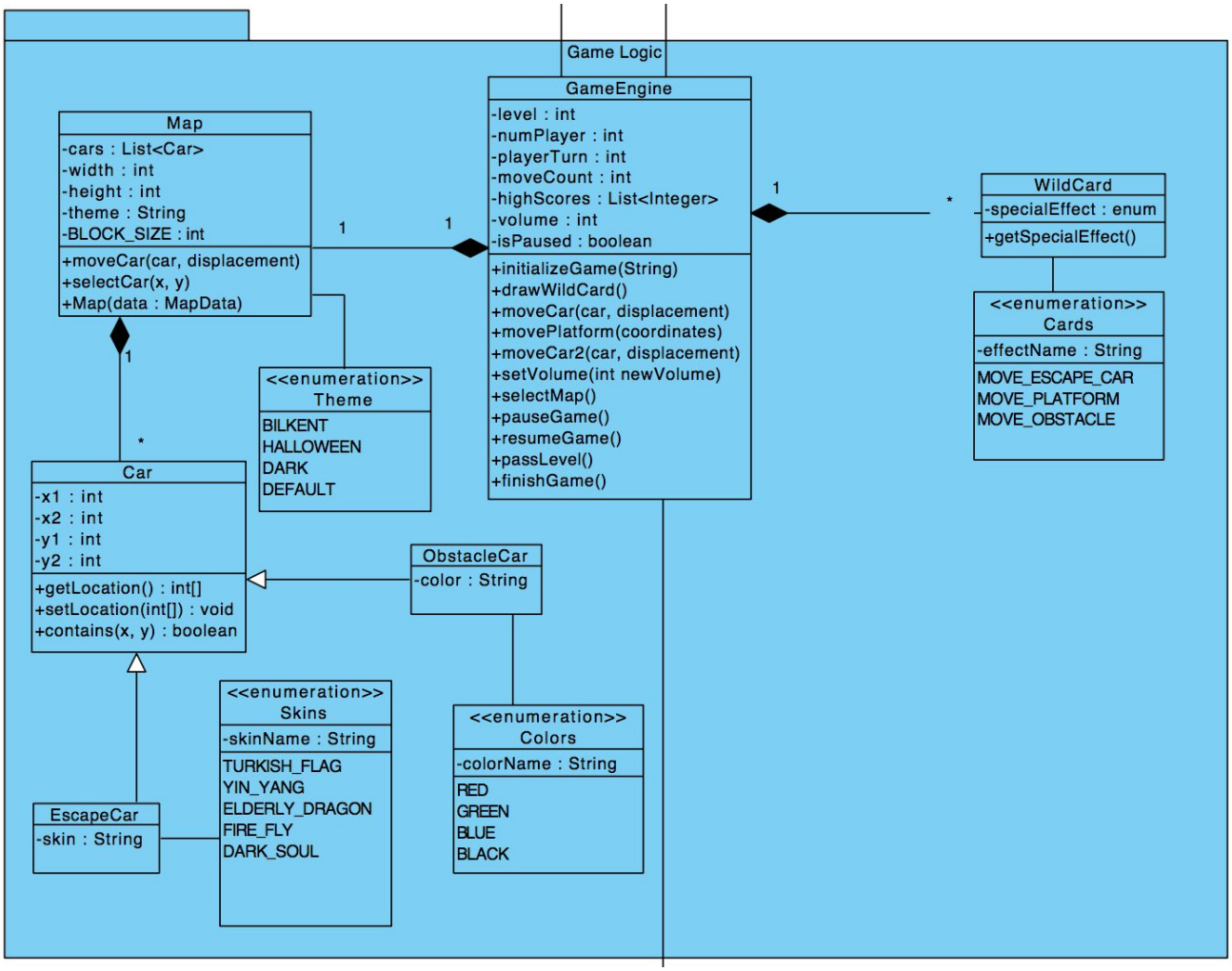


This class is the form of storage of Map data, that is stored in the application (MapDataManager class). The details of this class is given below.

Attributes:

- private List<Car> cars: The list of cars included in the map is stored in this list.
- private int width: The width of the map is stored as an integer.
- private int mapType: The type of the map is specified as an integer.
- private int height: The height of the map is stored as an integer.

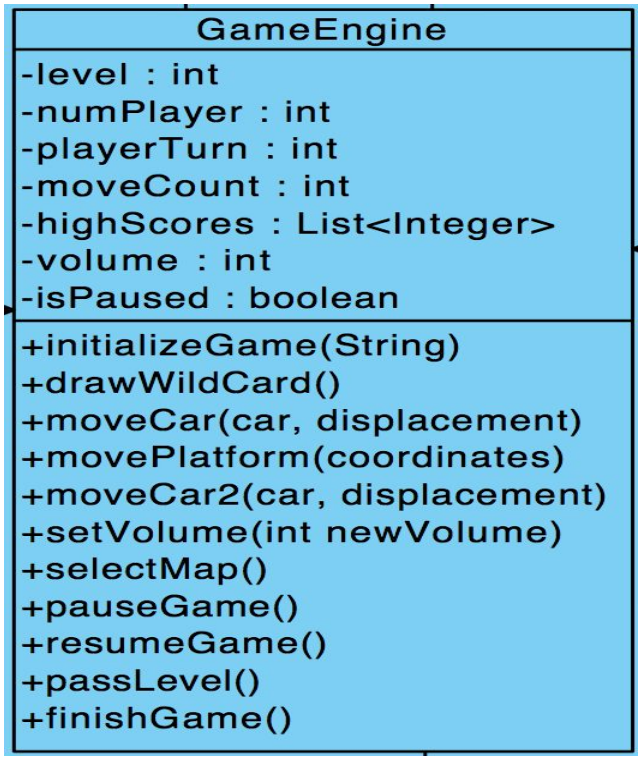
4.3.3 Game Logic Package



This is the package for the implementation of the logic of the game. This subsystem contains 4 crucial classes. These classes are GameEngine, Map, Car and WildCard classes.

GameEngine class performs the functionality of a manager class in order to game to be played. The defined functions are for the process of the game.
Map class defines all the required data for the initialization of the map and the current situation of the level. All of the user actions for playing the game are performed with the use of this class which is connected with the GameEngine class.
Car class defines the physical entity of a car in a virtual way. By providing the location information of the car instance the existence of the entity can easily be identified.
WildCard class is defining a special behavior to make a move that is only present in the multiplayer mode of the game. The special behavior is specified in the context of special effect.

4.3.4.1 GameEngine class



Attributes:

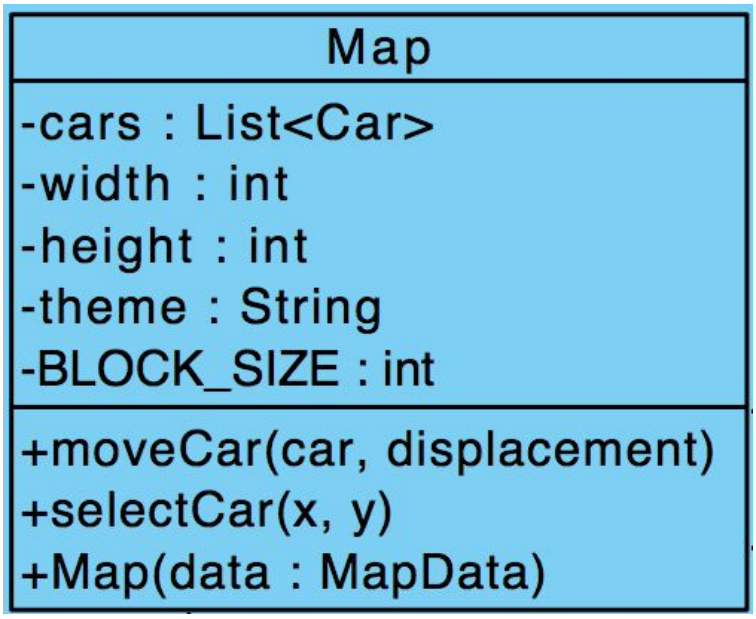
- private int level: The level that is being played is stored as an integer variable in this class.
- private int numPlayer: The number of players involved with the game is represented as an integer. This property defines whether the game is played in the Single Player Mode or Multiplayer Mode.
- private int playerTurn: The number of the turn that the player is performing is stored as an integer.
- private int moveCount: The count of the moves performed are stored as an integer.
- private List<Integer> highScores: The information regarding the high scores of the levels of the game is stored as a list of integers.
- private int volume:
- private boolean isPaused: The pause status of the game is stored as a boolean variable.

Methods:

- protected void initializeGame(String map): With the map information taken as a string, the specified level is initialized with this method

- protected void drawWildCard(): With this method a wild card to restrict the players' playing options is dealt to the player.
- protected void moveCar(Car car, int displacement): This method moves the first car in the desired displacement.
- protected void movePlatform(int[] coordinates): The game platform is moved to the desired coordinates with this method.
- protected void moveCar2(Car car, int displacement): This method moves the seconadry car in the desired displacement.
- protected void setVolume():
- protected void selectMap(): With this method the map selection can be done to play the game
- protected void pauseGame(): For the pause functionality, the game can be paused with this method.
- protected void resumeGame(): For a paused game, this method allows the user to resume the game.
- protected void passLevel(): As the level ends and the player successfully passes the level, this method enables the player to proceed to the next level.
- protected void finishGame(): This functionality allows to finish the gaming process and exits from the game.

4.3.4.2 Map Class



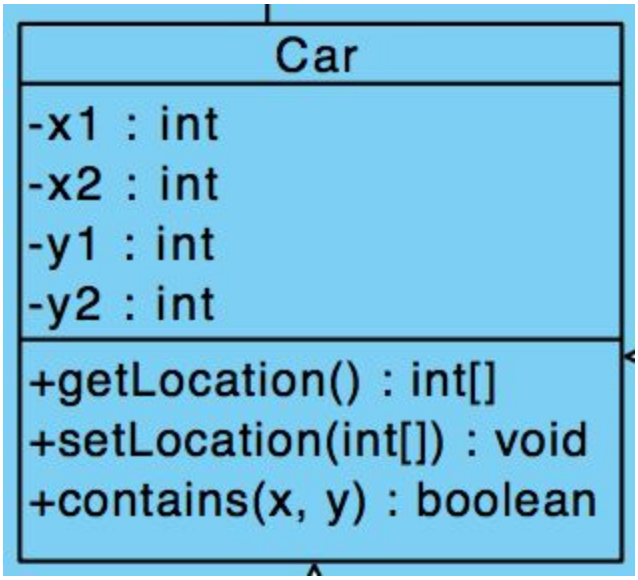
Attributes:
- private List<Car> cars: The cars present in the map are stored with this attribute
- private int width: The width of the map which the game is going to be played is specified as an integer

- private int height: The height of the map which the game is going to be played is specified as an integer
- private String theme: The theme which the map is going to be visualized is specified as a String
- final int BLOCK_SIZE: The size of a block in the board is specified as a constant integer.

Methods:
- protected void moveCar(Car car, int displacement): This method moves the desired car object in the desired displacement
- protected void selectCar(int x, int y): By a specific coordinate selection this method selects a car in the given coordinates.

Constructor:
- public Map(MapData data): With the provided MapData object as a source of data, Map with desired qualifications is constructed.

4.3.4.3 Car class



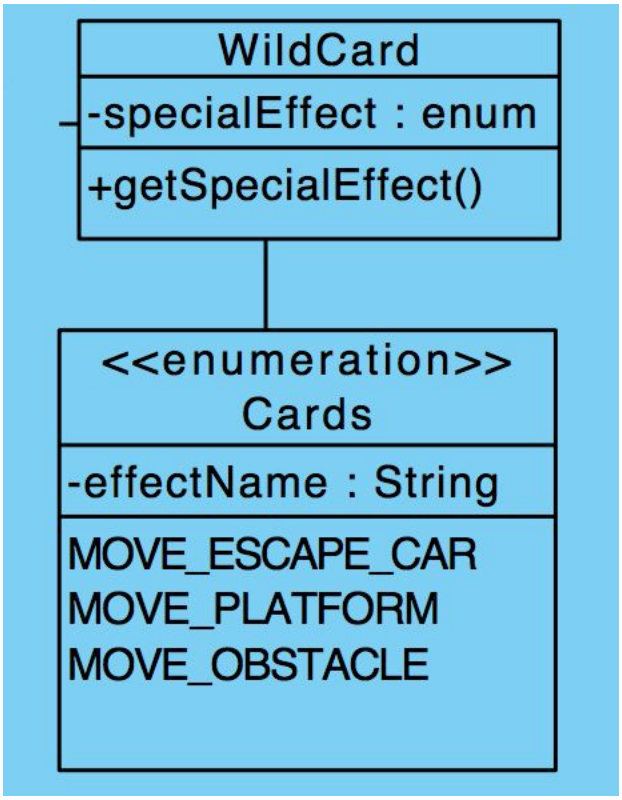
Attributes:
- private int x1: The x coordinate of one of two points stored for the car is stored as an integer
- private int x2: The x coordinate of the other of the two points for the car is stored as an integer
- private int y1: The y coordinate for the first of two point for the car is stored as an integer.

- private int y2: The y coordinate for the other of the two points for the car is stored as an integer.

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 conatins(int x, int y): This metod checks whether the coundaries of the car contains the given point (x,y). Returns true if the car contains the specified point.

4.3.4.4 WildCard class



Attributes:

- private enum specialEffect: The constraint that the wildcard is going to provide is specified by an enumeration that is stated as private.

Methods:

- protected String getSpecialEffect(): This method enables to access to the special effect specified in the wildcard. The special effect is returned as a string.

4.4 Class Interfaces

As stated in section 4.1, the class uses the serializable interface for enabling local data storage. In this motivation, High Score class is going to be implementing Serializable interface. Due to the requirement of the method signatures defined in the Serializable interface, the HighScore 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. Glossary & References

https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html
https://www.thinkfun.com/products/rush-hour/
https://www.thinkfun.com/products/rush-hour-shift/