CS 319 - Object-Oriented Software Engineering

System Design Report – II

Rush Hour

Gurup Şurup

Muhammet Said Demir

Ata Coşkun

Zeynep Nur Öztürk

Asuman Aydın

Tarık Emin Kaplan

Supervisor: Eray Tüzün

TA: **Muhammed Çavuşoğlu**

Table of Contents

1. Introduction

1.1. Purpose of the system

Rush Hour is a 2-D parking lot, escape and strategy game. The implementation of the game is simplistic where the themes and the sounds of the game are also increasing the fun one can get from the game. It is easy to play and also user-friendly in a way that user can get a hint in the game when one is stuck with the level. Also, it has specific qualities from differentiating it from the original game. For instance, one of them is obstacles where pushes to the player to think more and have more complex. Player's goal is to take the specific car out from the parking lot where there are other cars in different shapes and sizes. By increasing the challenge of the levels, the game contributes a joy to the user. To give diversity to the game, there is a multiplayer option and the functionalities of the multiplayer part vary from the single player one in every sense. So, the game is mostly easy to play, user-friendly and also enjoyable with its different themes and different game modes.

1.2. Design goals

In this state, design allocates the big amount of place in the system with respect to implementation. For this game, the design includes mostly the user interface and how the game is played from the developer point of view. Also, the non-functional requirements are reviewed and had better understanding according to the implementation of functional requirements. So, the design basically makes the job of implementation easier in this sense. To clarify the non-functional requirements, the descriptions are in the following chapters.

1.2.1. Trade-Offs

Development Time vs. Performance:

The first thought of the implementation was the usage of Unity. Even though it seemed to implement easier with Unity, we had a choice by side of Java for the simplicity of the development. To control the implementation and increase the development, later on, Java is more developer friendly. Although Java will give more freedom to update later, this reduced the performance because Unity has better performance with running the engine and concatenating with the User Interface.

Memory vs. Maintainability

Classes and the attributes are the main parts of the implementation. While designing, these specifications make us understand the design better. However, for the memory usage, it was more taken into consideration those rather using objects to connect method and attributes than using multi-leveled arrays. By this way, the memory allocation would be decreased. Across this, the maintainability is placed with the object-oriented design. The objects and classes are used with inheriting each other.

Understandability vs. Functionality:

Every difference we make in the design is aim to have more understandable and not completed implementation. To increase the usability, we also prefer the understandability of the game over functionality. While talking about the functionality, we mean complicated instructions and complex usage of the keyboard while controlling the game. Since the game doesn’t include a lot of control management, it simply focuses on the understandability of the game.

Usability vs. Functionality:

Since our game is mostly aiming to give better time with the good interface and there are no complicated instructions to play the game, we choose the usability over functionality. There is not multi-screen to play the game, so this also represents the preference the usability over functionality. Our game is easy to understand and it is user-friendly since it has the how to play a part on the main menu to increase the usability for the player.

Space vs. Speed:

The focus for the game engine is more on a strategy of the game logic. So, we rather used the specific property for the red car than separating it from other cars. While this increases the memory to reach and differentiate the specific car from the others, the time to find the variable and to execute the calculations decreased respectively.

Development Time vs. User Experience:

Although the first thought was using the Unity to implement our game, we had the change with Java. This choice increased the development time because while Unity has the asset packages to use in user-interface and it is fast to apply to the game implementation, Java leaves the design and implementation mostly to the developer. However, creating a new design and considering the user's perspective, our game will be simpler with a user-friendly interface. No complex sight and struggles to understand the game will be allowed.

1.2.2. Criteria

End User Criteria

Usability:

For the ease of playing the Rush Hour, the design is simple and user-friendly made. It has a minimal menu and also there are themes. Another issue is figuring out how to play the game. In this point, the game has a feature which can be reached from both in the game and

in the main menu. It basically includes a video that displays the main instructions for the game. Although there is a help option, the game is easy to understand with its UI. The specific car is distinguished from other cars with its color and shape. So, the player can simply say what he/she has to do.

Performance:

The importance of the performance appears where the theme is changing and also when the user wants to continue the game where she/he left. We were ready to collaborate with the Unity packages but change on implementation pushed us to use Java. Without looking at it from the lost sides, we can just say that GUI library of Java and our design now have the simpler performance.

Maintenance Criteria

Extendibility:

As the implementation part depends on the design, the features and functionalities of the game are set with an abstract way and then implemented. However, according to the player views, we should be able to change the game. For this reason, the game should be opened to changes and, in the future, it should have space and opportunity to extend the game such as adding or changing the level or themes or adding a new type of multiplayer game or resume button (see 2.3).

Modifiability:

The systems in implementation are depending on each other. However, there can be still changes in the implementation so the systems and their subsystems do not get affected by each other's changes. For the help of object-oriented programming, the instances and other functions attributes may change but do not affect the way the game is played by the player.

Reusability:

Even the systems are connected; there are points in the implementation that are not connected at all to the game. The most of this part is basically the GUI part. With the system separations in the implementation, changing one part of the system will not affect that much of the rest of the implementation. Also, different systems allow us to reuse parts of it if we want to update the game.

Portability:

The significant merger of the properties and criteria of the game is supplied with the portability. As we implement our game with Java, it provides us a freedom to have our game implemented in different portals with its JVM.

Performance Criteria

Response Time:

The concept of the game is to be fast enough to catch up with the user reflexes and also to have enough joy from the UI. So, the response time is a big issue to handle an implementation part. Also, the animations over the car movements and obstacles are direct so that the game appears smoother.

1.3. Definitions

Java Virtual Machine (JVM) – a machine which allows computers to run Java

Model View Controller (MVC) – Updating, appearance controlling tool for Java.

Graphic User Interface (GUI) – Total Frame classes for Java.

2. System Architecture

In this section, our system's overall architecture and the key structural decisions we have agreed upon will be explained, pointing to the main structural necessities of a program, which are (in order of appearance in the report): subsystem decomposition, hardware/software mapping, persistent data management, access control and security, and boundary conditions; emphasis being on the subsystem decomposition part.

2.1 Subsystem Decomposition

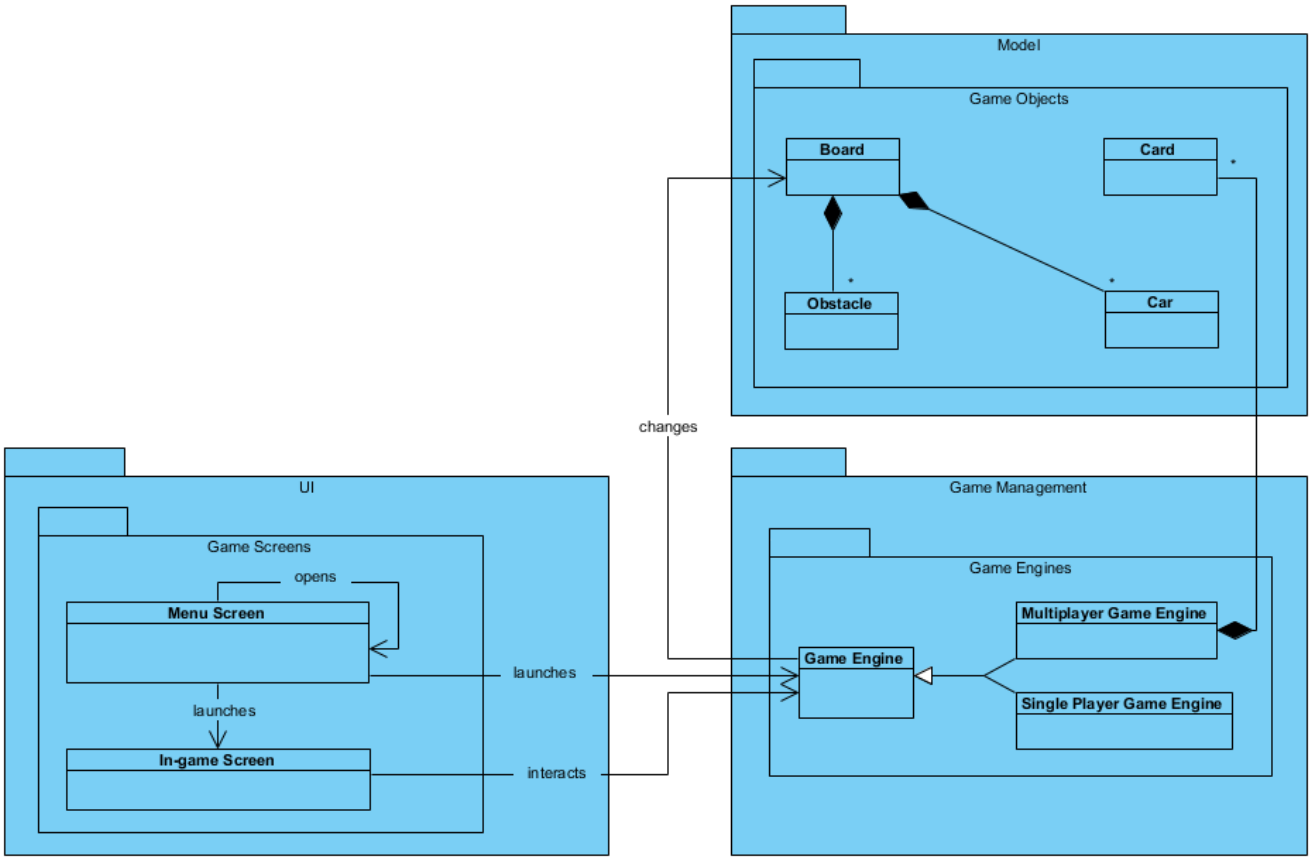


Figure 1: Subsystem Decomposition Diagram

As you can most likely already guess from the diagram given above (Figure 1), we have decided to design our program with a MVC (Model-View-Controller) design pattern, considering it was applicable and most fitting in our case.

It was easy to match our system with the 3 subsystems MVC proposes:

- The Model subsystem, self-explanatorily, corresponds to Model,
- The UI subsystem corresponds to the View,
- Game Management subsystem corresponds to the Controller structure of MVC model

Both MVC and our game are, in a non-degrading way, simple (compared to other design patterns for MVC; or compared to a more “real” product for our game), making MVC a perfect fit for our game.

On top of that, it is easy to represent the relations between classes and subsystems with MVC, and those relations will be explained just on the following lines.

Just by taking a quick look at our subsystem decomposition diagram (Figure 1), you can already see the skeleton of MVC design pattern which can be expressed as: User uses controller, controller manipulates model, model updates view, user sees view, and continues on using the controller, and so the cycle goes on.

In our case, a user first comes in contact with a main menu screen (which is a UI/View element, which is a game screen, and a menu screen), from there the user can navigate to other “View” type of elements, which are other game screens. A menu screen may open other game screens (such as main menu screen opening a game mode selection screen when user clicks on “Play”), when user navigates to a game screen which hosts an actual game (for example, user chooses a level on a single player mode level selection screen), an in-game screen, and a game engine corresponding to the chosen game mode (Single player/Multiplayer) is created.

A game engine belongs to the Game management subsystem which corresponds to Controller, and basically controls a game, by interacting with an in-game screen (view) and a board (model) simultaneously.

A board exists in any type of game mode and is basically where all the game objects play on, so by interacting with a board, a game engine can interact with game objects that belong to that board as well (such as moving cars, or initializing obstacles at the start of the game; board can also be moved in multiplayer mode too).

A card holds information about what a player can do in his turn in multiplayer mode. This will be explained in detail in following sections.

We believe this design that we ended up with has optimal cohesion and is loosely coupled, making our game extendable and flexible.

2.2. Hardware/Software Mapping

Rush Hour game will be implemented in Java. Initially we first decided to do it on Unity, and using C# for our scripts, but discarded that idea due to design concerns, specifically, we were afraid we couldn't be able to release a product where either 1- there wouldn't be too much code, and it would look like we didn't do much, it was purely Unity's work; or 2- the end product wouldn't be satisfactory to meet the expectations.

So, we decided to head back to the domain we already were comfortable in, which is Java, and decided to make the game desktop only. In terms of software, our game will require Java run-time environment and/or Java development kit (8 or higher). As such, the software and hardware requirements are the same as for running any other Java 8 program

Windows

- Windows 10 (8u51 and above)
- Windows 8.x (Desktop)

- Windows 7 SP1
- Windows Vista SP2
- Windows Server 2008 R2 SP1 (64-bit)
- Windows Server 2012 and 2012 R2 (64-bit)
- RAM: 128 MB
- Disk space: 124 MB for JRE; 2 MB for Java Update
- Processor: Minimum Pentium 2 266 MHz processor

Mac OS X

- Intel-based Mac running Mac OS X 10.8.3+, 10.9+
- Administrator privileges for installation

Linux

- Oracle Linux 5.5+1
- Oracle Linux 6.x (32-bit), 6.x (64-bit)2
- Oracle Linux 7.x (64-bit)2 (8u20 and above)
- Red Hat Enterprise Linux 5.5+1, 6.x (32-bit), 6.x (64-bit)2
- Red Hat Enterprise Linux 7.x (64-bit)2 (8u20 and above)
- Suse Linux Enterprise Server 10 SP2+, 11.x
- Suse Linux Enterprise Server 12.x (64-bit)2 (8u31 and above)
- Ubuntu Linux 12.04 LTS, 13.x
- Ubuntu Linux 14.x (8u25 and above)
- Ubuntu Linux 15.04 (8u45 and above)
- Ubuntu Linux 15.10 (8u65 and above)

As an unlisted hardware requirement, any functioning mouse will be needed to satisfy the input needs of the game. Player will do all kind of interactions such as moving the cars in game, or interacting with the options in the menu or settings screens, by interacting with a mouse.

2.3. Persistent Data Management

All the game objects will be initialized as the game starts and will be deleted as the game is closed, meaning they will be up as long as the game instance is running on the system. Other than that, the player's personal settings information, high scores and his progression in the game will be stored in a text file. To store our assets we will use in game we will use .png and .gif formats, and to store the sound files we will use .waw format.

2.4. Access Control and Security

The multiplayer in game will be in a local multiplayer setting, meaning both the players will play on the same phone / PC simultaneously, and there won't be any sort of login/sign-up type of storing player information, so there won't be any need for security concerns. This also means that the player high scores and level progression will be localized as well, meaning the player needs to start over if they choose to play the game on a different device.

2.5. Boundary Conditions

Rush hour does not require any installation because it will come as a .jar file. Since we intend to make this file available to download from the internet and into your device, there should be no need to transfer the game from one device to another, increasing accessibility. Rush hour can be terminated by choosing the exit option in the main menu or in the in-game settings screen, or by pressing the "X" on the top right of the window.

If for any reason the game crashes, the player's current progression, scores and personalized settings might be lost. Any sort of corrupted data should result in turning to initial state of the game when it was first installed (for example, a corrupted setting data will result in settings being set to default).

3. Subsystem Services

3.1. UI Subsystem

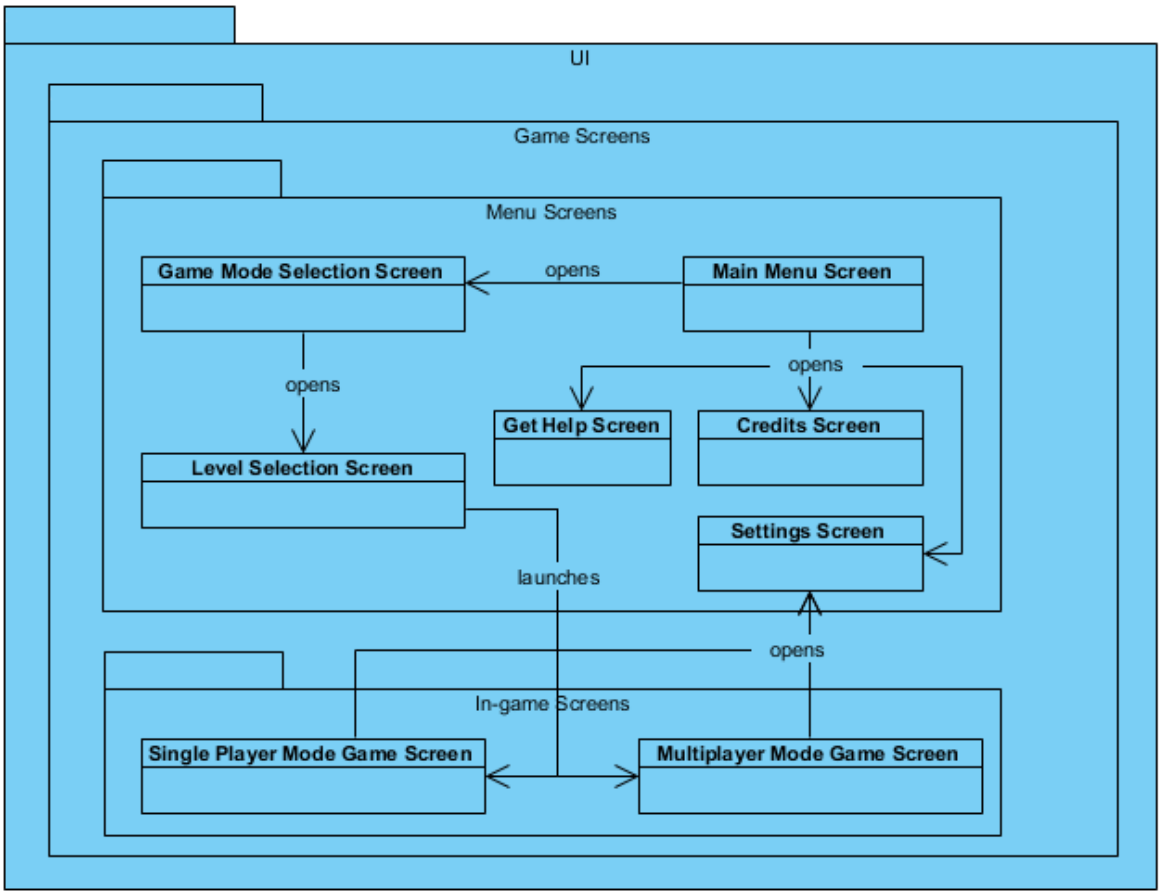


Figure 2: UI Subsystem in detail

UI subsystem is responsible for creating screens according to the user input. It mainly has 2 Game screen elements, which are Menu screens and In-game Screens.

There is no need for a communication between subsystems for navigating between the menu screens. Menu screens consist of the main menu screen the user is initially presented when the game is launched. From there, the user can access the settings screen where the user can customize the game settings to their preference, the credits screen where they can view the credits, get help screen where they can learn the basics on how to play the game,

the game mode selection screen where the user selects whichever game mode they want to play, and from there, the level selection screen where, according to their prior game mode choice, the user is presented with a screen consisting of levels to choose from.

When a level is chosen from any game mode, it's corresponding in-game screen is launched, and only then a communication between said level selection screen and the game engine occurs. Based on user's game mode and level selection, a game engine is created, and this game engine creates the initial state of board and its game objects, and shows them in the already allocated in-game screen. The user can also open settings screen from an in-game screen as well.

3.2. Game Management Subsystem

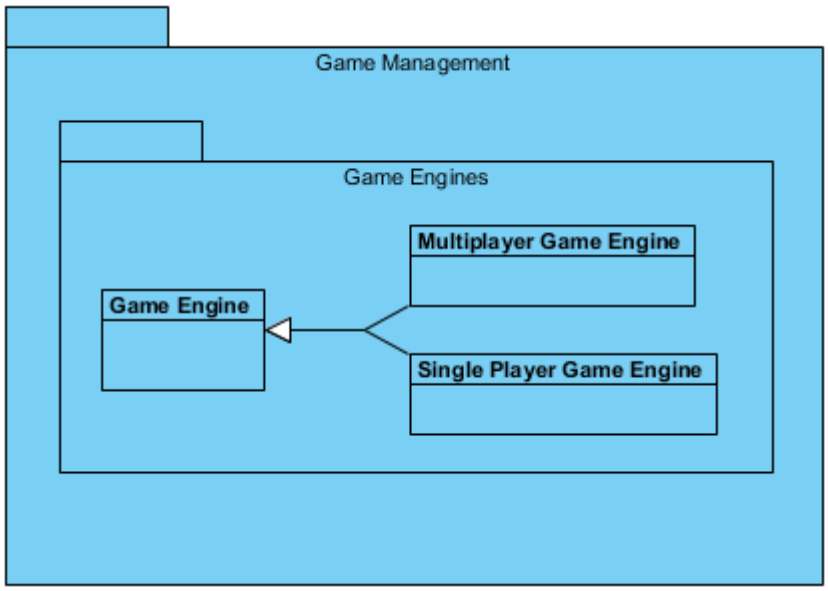


Figure 3: Game Management Subsystem

Game Management Subsystem creates a game engine with respect to the mode selection of the user, and a game engine controls the game by initially setting the game objects on an in-

game screen (view) at first, and upon user's commands that come through information from that same in-game screen, interacts with game objects, from Model subsystem.

Both Single Player and Multiplayer game engines share some common aspects, so they are connected to a single generalized game engine class. However, there are also big major differences, so merging them on only one game engine class is not a good idea. These differences and similarities will be shown more in detail in the upcoming Low-level design part of this document.

3.3 Model Subsystem

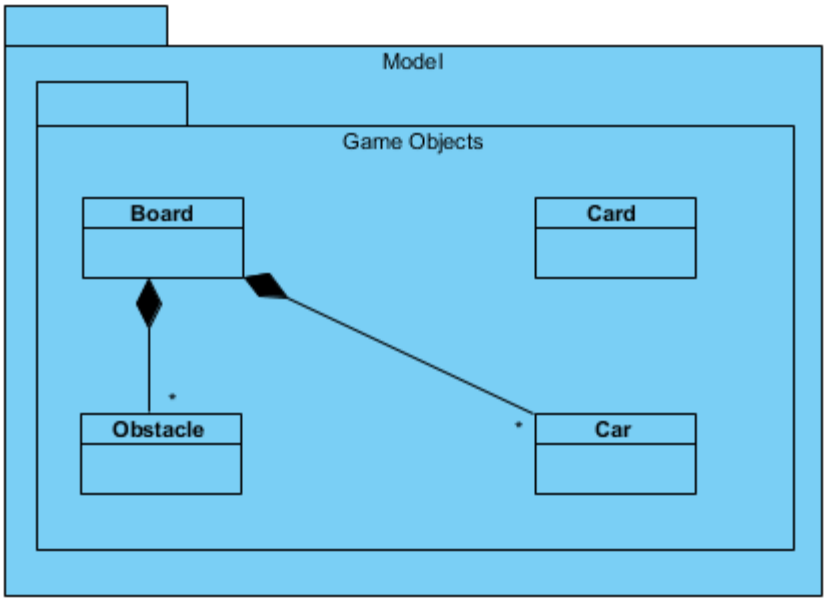


Figure 4: Model Subsystem

Model Subsystem holds the game objects which are interacted with throughout the game. A board exists in any type of game mode, and it is tile-mapped, meaning it has unit locations on it that can be occupied by other, non-board game objects.

The single player board is a simple 6x6 square board and can't be moved, but the multiplayer board is a 14x6 board that consists of two 5x6 board parts on left and right and

one 4x6 board part in the middle, the interesting thing about the multiplayer board is that the two parts on the sides can be moved vertically to impact the game in a whole different way.

As mentioned before in part 2.1, a board consists of and is basically where all the game objects play on, so by interacting with a board, a game engine can interact with game objects that belong to that board as well. Meaning that, in order to move a car, user clicks on the car in the in-game screen (view), and drags that car to some other point, game engine (controller) checks whether this move is legal and is applicable on the current state of the board (model), and updates the current state of the board accordingly and displays that change (that is, if any change has been made) on the in-game screen.

Obviously the most interacted game object would be cars (which are either of 1x2 or 1x3 size), but there are other game objects that a Board may have too, which are obstacles, 1x1 sized objects that occupy a place on the board and no car can pass through them.

The playing mechanisms of the single player mode and multiplayer mode is fundamentally very different (this is also another reason why there is two different game engines). On single player mode, the only actor interacting with the game is a single person, as such, there is no constraint on them other than time to interact with a single player mode level; however, since there are two different players interacting with the game on multiplayer mode, the game is played in a turn based manner, and a constraint regarding playing their turns for both actors is necessary, and that's where the Cards come in to work. Cards are game objects unique to the multiplayer game mode, and they hold information about what can a player do if they decide to play that card on their turn (like number of moves that a player can do that turn or whether they can shift the board or not). Both players initially start with multiple cards, they play a card, then do whatever move they want on the board

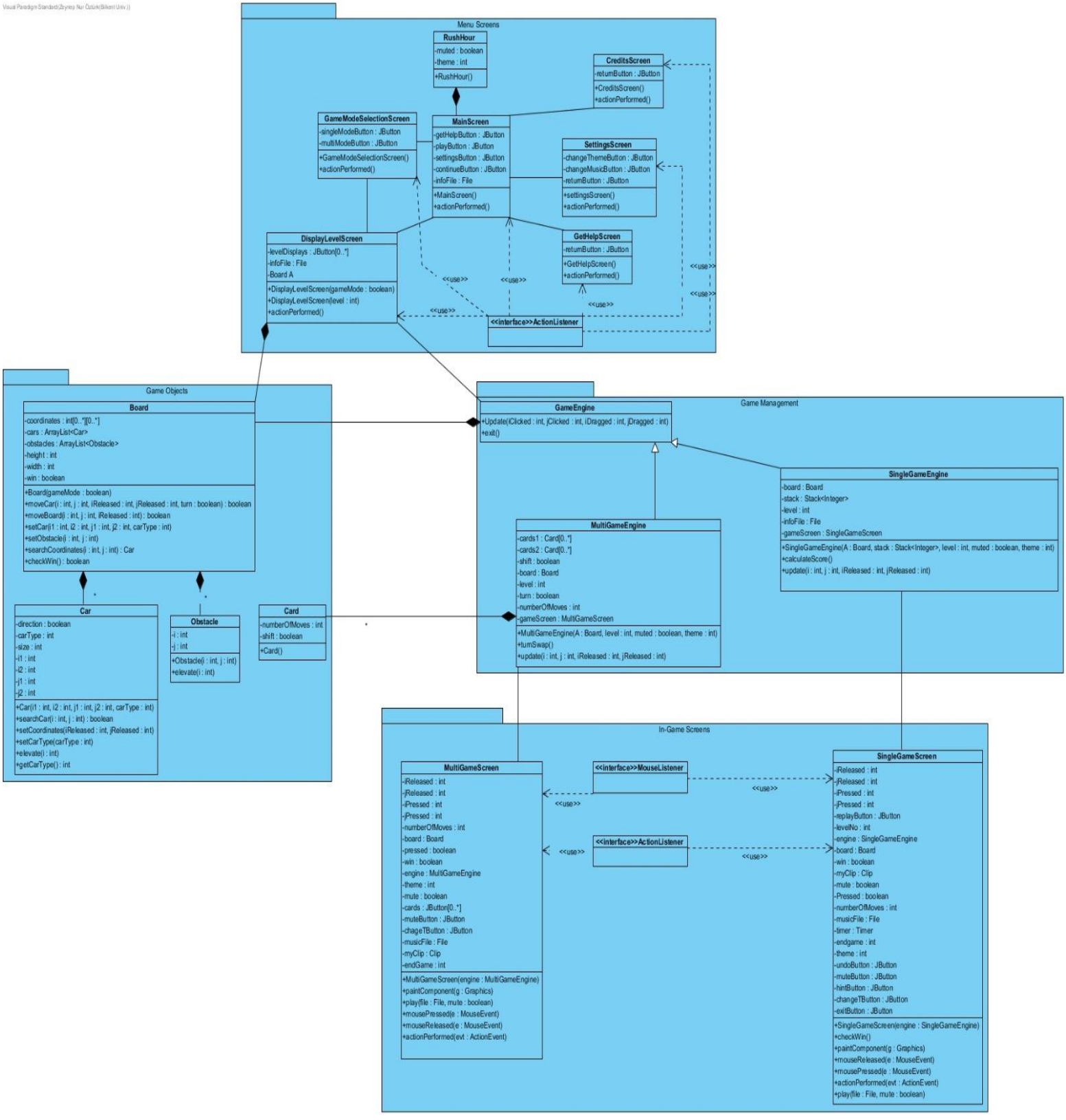
as long as their played card allows it, then their turn passes to the other player. At the start of the turn, a player also draws a new card. Since both Board, Cars and Obstacles are game objects that a player has to interact with directly, they are linked with each other; but Card is a model that holds information about what can be done that turn, player effects the game through them, so the Cards indirectly affect the game; they are also unique to the multiplayer game mode, for these reasons, Card is not bound to the other game objects, but rather the multiplayer game engine (this relationship can be seen in Figure 1 in section 2.1).
As we have explained in our analysis report, we also intend to add two new features to the game, portals and bridges, both of them will be available to use in multiplayer game mode. Portals are in pairs, meaning there are always an even number of portals in a level, and one pair of a portal only works with the other. When a car comes into interaction with a portal, it is teleported to the other pair of portals (if the board state at that moment allows it, meaning there isn't anything to block that car on the "destination" portal). Portals are mainly used to change the horizontal/vertical setting of a car, because there is no other way to change it (a car can only move on its initial horizontal/vertical setting, as long as it doesn't go through a portal).
The bridges are 3x1 sized, limited-use tools, that enable a car (with horizontal setting) to pass over it and another (with vertical setting) to pass under it. The bridge can be placed on an empty 3x1 space, or on top of an obstacle (which is of 1x1 size), or on top of a car (with vertical setting) as long as the car is below the middle portion of the bridge, (the part that allows vertical movement).
Since both portals and bridges are not implemented in the game yet, we didn't show them in the diagram, but since they are also game objects that user directly interact with to affect

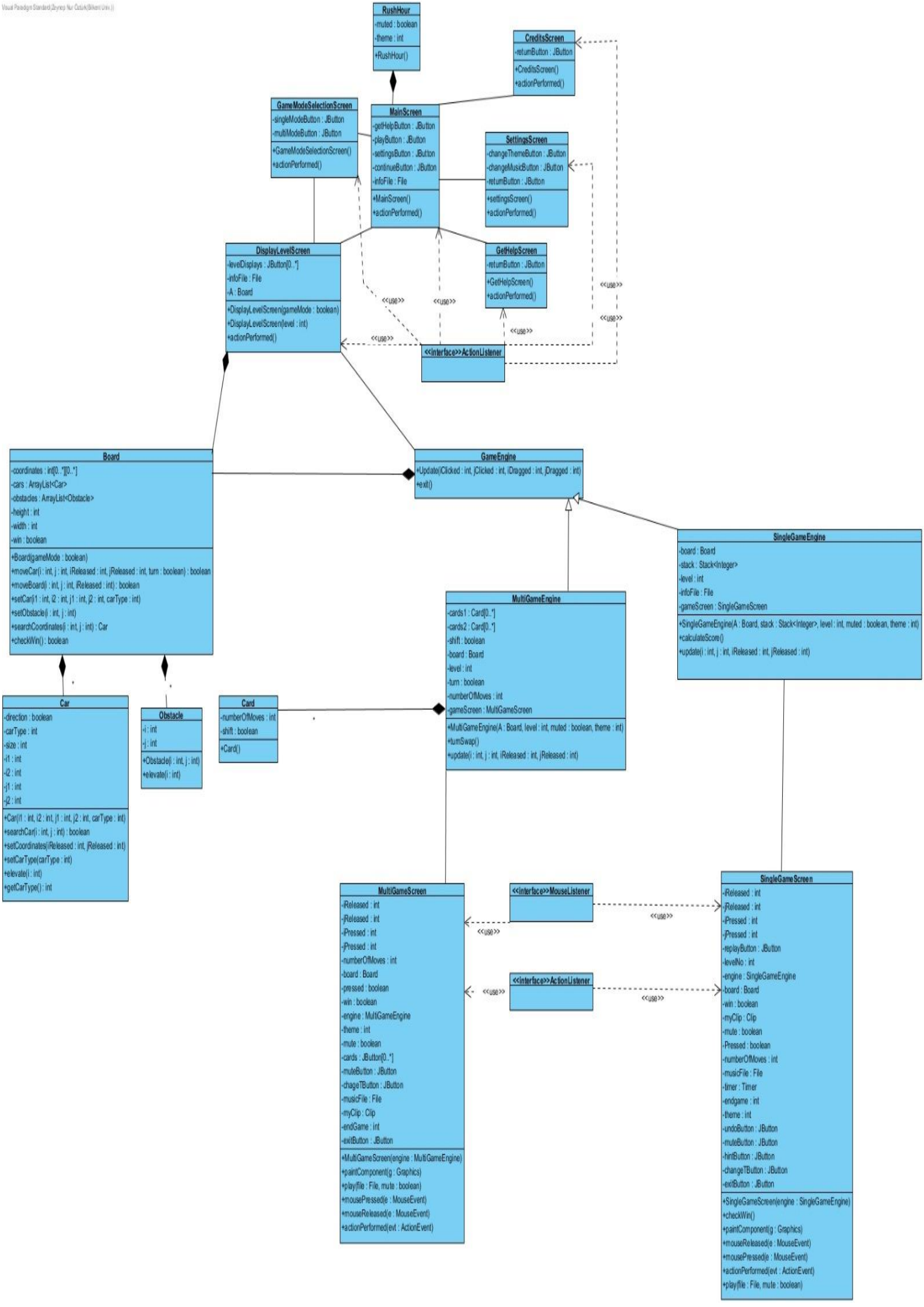
the game, you can expect both of them to be in a “has a” relationship with Board, just like Car and Obstacle does.

4. Low-level Design

4.1. Final Object Design

We will give the full diagram, then divide into pieces





4.2. Packages

In our implementation we have used two different kind of packages, the first type is external libraries packages and the package that we defined.

4.3.1. Package Introduced by Developers

4.3.1.1. Menu Screens Package

This package includes the classes which are responsible for representation of the game menus and their necessary methods. Until the game screen opens user sees this package's screen.

4.3.1.2. Game Management Package

This package includes the classes which are responsible for the managements of the game. Game engine for the single and multiplayer modes are managing from those classes.

4.3.1.3 Game Objects Package

This package includes the objects of the game. The objects which are used in game.

4.3.1.4 In-Game Screens Package

This package includes the screens when the user is in the game. The real game is played in those screens by the user.

4.3.2 External Library Packages

4.3.2.1. java.util

This package contains frameworks, event model, time facilities, random number generator, stack, arrays and array lists. We use array lists to control the objects in our game like cars, obstacles; arrays for our board's coordinates; stack for our undo method.

4.3.2.2. java.awt

This package contains all the classes to create a user interface and for the painting graphics and images. Therefore, we use it to draw our game's user interface.

4.3.2.3 java.io.

This package used to provide system input and output through data streams and file systems. Because that we are using the file system to get the game's levels and other things, we used this package

4.3.2.4. javax.imageio

This package is used for images inputs and outputs in our game.

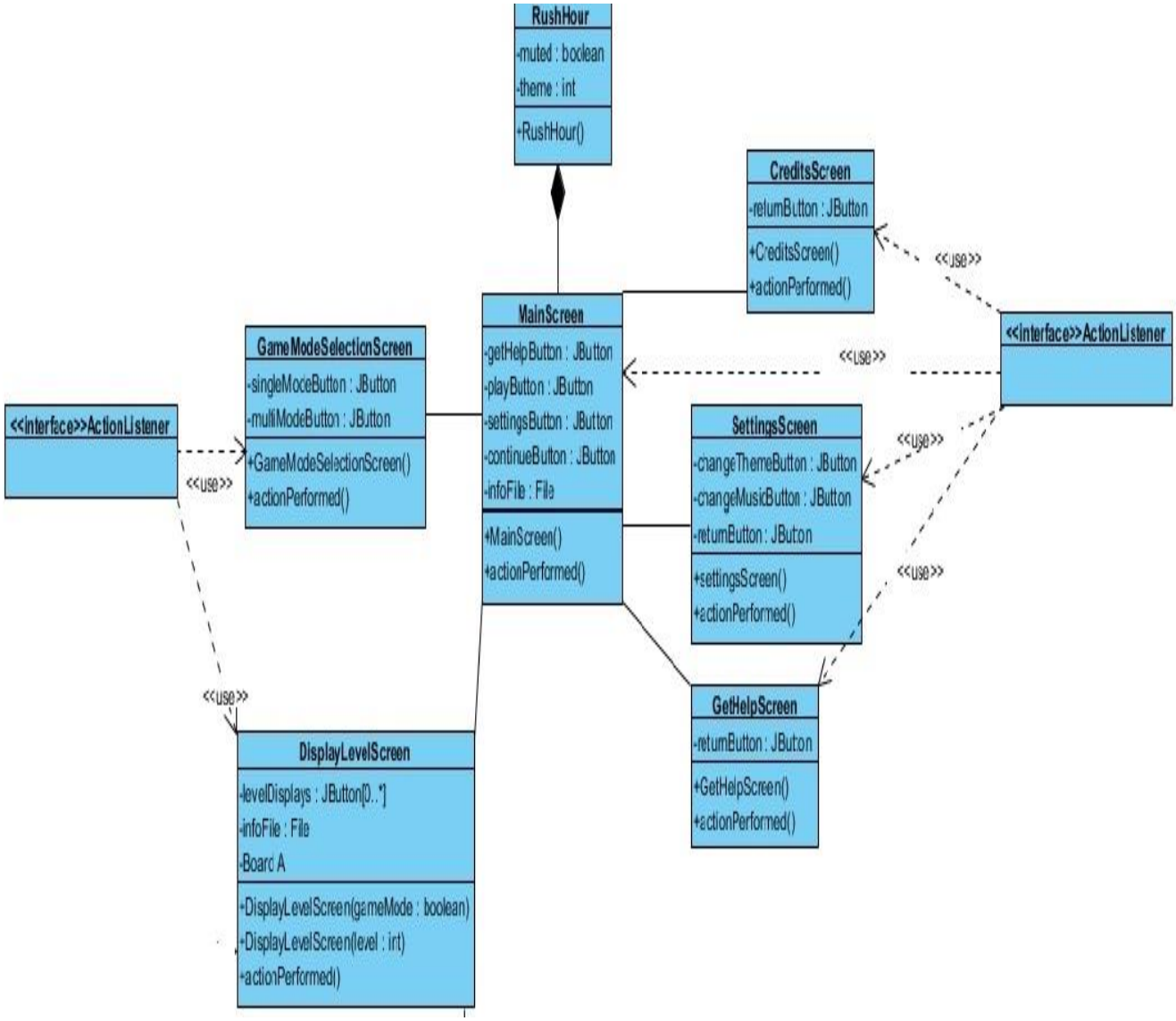
4.3.2.5. javax.swing

This package is used for our main GUI. This gives us all the frames, buttons, and so on. Our main menu screens and in-game screens are based on this package.

4.3. Class Interfaces

In this section, we have introduced our class interfaced more detailed and clearer for any implementer to code it more efficiently.

Piece 1, Menu Screens:



Menu screens are basically what you see as you start the game until you start single or multi game mode. It consists of several screens which are connected together.

4.3.1. RushHour Class

This class is the main class which will be start to run as the first thing that runs in this program.

Attributes:

muted : boolean: The initial voice level of the program. The player can mute the game from SettingsScreen Class and when they do it, the game will be start as muted. This boolean makes that happen.

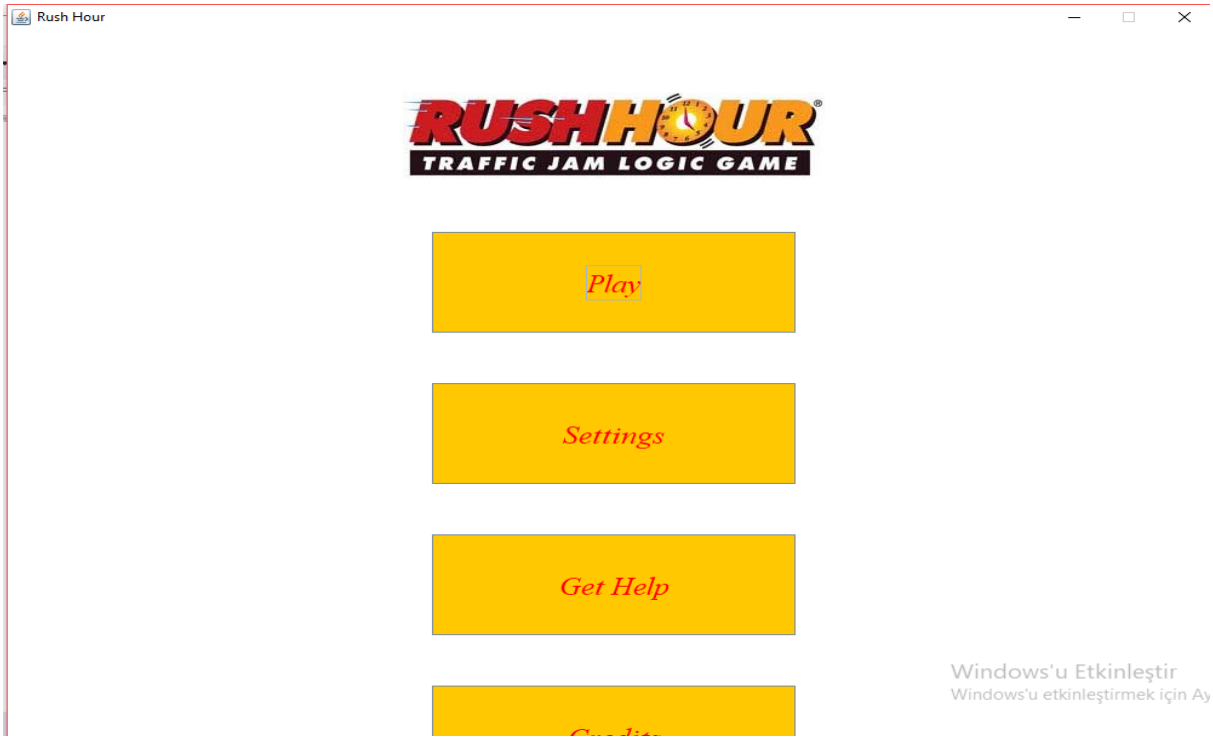
theme : int: The initial theme preference. Same as the muted attribute, this can also change from the SettingsScreen Class.

Constructor:

RushHour(): It calls the MainScreen Class

4.3.2. MainScreen Class

This class is the main screen that will show basically this:



The credits button is half visible here, and also there will be a resume button as well, each button will redirect the player to another class.

Attributes:

getHelpButton : JButton: Button that will redirect the player to the getHelpScreen.

playButton : JButton: Button that will redirect the player to the gameModeSelectionScreen.

settingsButton : JButton: Button that will redirect the player to the settingsScreen.

continueButton : JButton: Button that will redirect the player to the last played single game level, which will basically take the info from an infoFile and call the second type of constructor of DisplayLevelScreen Class with that info.

infoFile : File: From this file we will take the last played level info.

Constructor:

MainScreen(): It sets the GUI of the buttons and images and gives them an action listener.

Methods:

actionPerformed(e : ActionEvent): Action performed controls the buttons. If players pressed any buttons, this method handles appropriate changes.

4.3.3. SettingsScreen Class

Player can reach the settings from the main menu and start to play with those initial settings

Attributes:

changeMusicButton : JButton: Button that will mute/unmute, so that the game will start with this voice setting.

changeThemeButton : JButton: Button that will change the theme, so that the game will start with this theme setting.

returnButton : JButton: Button that will redirect the player back to the MainScreen.

Constructor:

SettingsScreen(): It sets the GUI elements and gives them an action listener

Methods:

actionPerformed(e : ActionEvent): Action performed controls the buttons. If players pressed any buttons, this method handles appropriate changes.

4.3.4. GetHelpScreen Class

It shows couple of photos that was take by us from the actual game's at Bilkent Station manual. They basically show how to play.

Attributes:

returnButton : JButton: Button that will redirect the player back to the MainScreen.

Constructor:

SettingsScreen(): It sets the GUI elements

Methods:

actionPerformed(e : ActionEvent): Action performed controls the buttons. If players pressed any buttons, this method handles appropriate changes.

4.3.5. CreditsScreen Class

It shows couple of photos and names of our group and our TA and instructor.

Attributes:

returnButton : JButton: Button that will redirect the player back to the MainScreen.

Constructor:

CreditsScreen(): It sets the GUI elements

Methods:

actionPerformed(e : ActionEvent): Action performed controls the buttons. If players pressed any buttons, this method handles appropriate changes.

4.3.6. GameModeSelectionScreen Class

It shows two buttons that make the player choose which mode he/she wants to play, multi or single.

Attributes:

singleModeButton : JButton: Button that will call the DisplayLevelScreen class with single-mode selected information.

singleModeButton : JButton: Button that will call the DisplayLevelScreen class with multi-mode selected information.

Constructor:

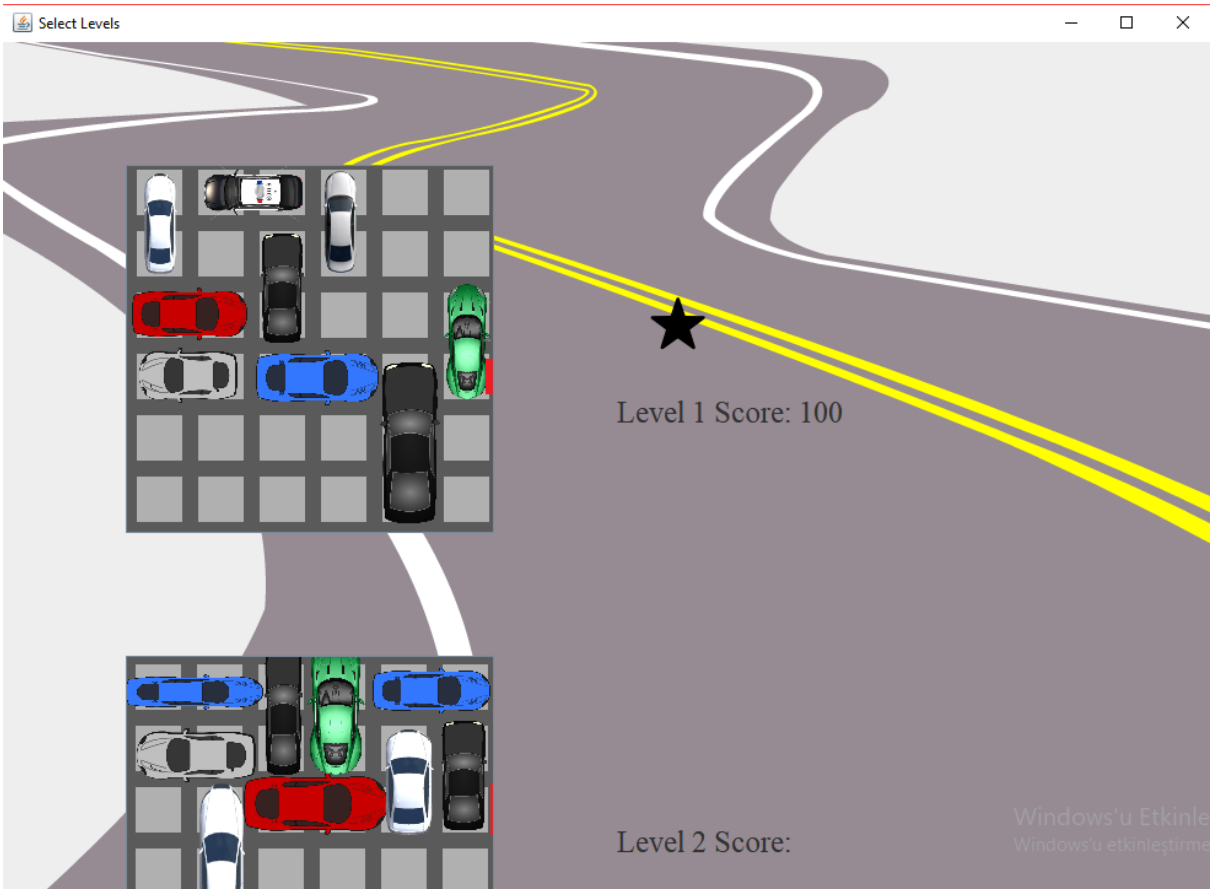
GameModeSelectionScreen(): It sets the GUI elements

Methods:

actionPerformed(e : ActionEvent): Action performed controls the buttons. If players pressed any buttons, this method handles appropriate changes.

4.3.7. DisplayLevelScreen Class

This class holds the hard-coded information of the levels and shows a list of them like this:



High scores are taken from the infoFile. This class creates a board object and gives the information of the chosen level's cars and obstacles to that board and calls either the multi or single game engines with the chosen level information and that board.

Attributes:

levelDisplaysButton : JButton[0..*]: Button array that shows a basic description of that level, level's name and high score of that level.

infoFile : File: File that will give the high score informations.

A : Board: Board that will be filled with the information of the chosen level's cars and obstacles. (and maybe the portal's information, if we'd implement that)

Constructor:

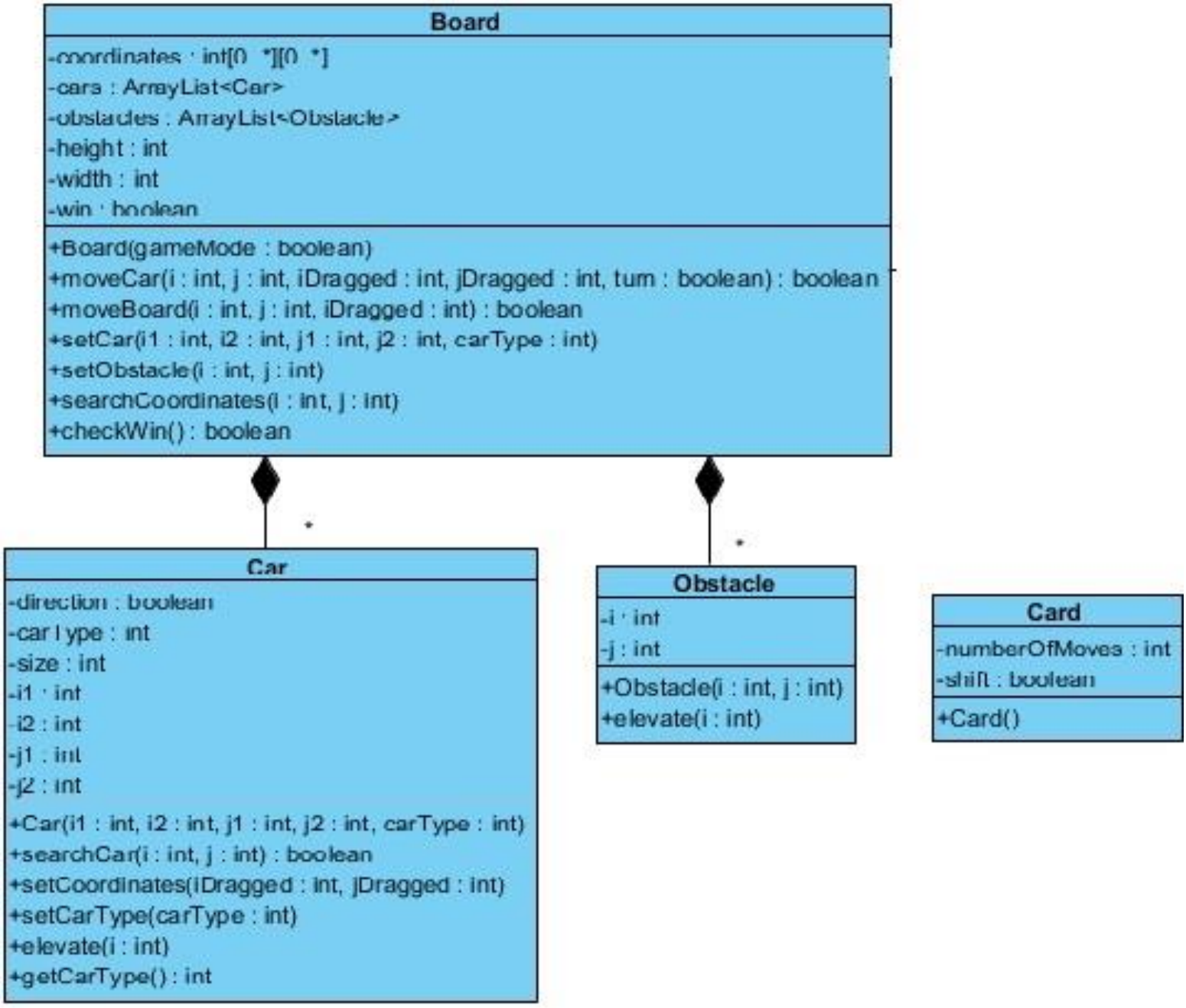
GameModeSelectionScreen(gameMode : boolean): It displays the level buttons according to the game mode. If it's false, it will display single levels, and vice versa.

GameModeSelectionScreen(level : int): It acts like that level is already chosen and do the rest of the things, like setting the initial board and calling the SingleGameEngine. This in only the case of resume button is pressed in the main menu, which will work only for single mode.

Methods:

actionPerformed(e : ActionEvent): Action performed controls the buttons. If players pressed any buttons, this method handles appropriate changes.

Piece 2, Game Objects:



Game Objects Subsystem consists of four classes which store the value about the game objects. They will be the classes that the game engine will control and change depending on the situation. The main objects of the game are cars and obstacles that are placed on the board. In the name of object-oriented programming, we've decided that the game engine can only reach the board instance which is given by level selecting class. And the only board will be able to reach the car and obstacle lists which are an attribute of this class. Furthermore, we tried to keep public function number as low as possible so that an outside function cannot reach to all of the functions and attributes of another class. Also, our fourth object is the Card object. In the multiplayer game, we have 10 different cards. We decided that only MultiGameEngine can reach the card to distribute random cards to the users.

4.3.8. Board Class

Attributes:

coordinates : int[0..*][0..*]: This attribute is used to display a gridded board as a double integer array

car : ArrayList <Car>: This attribute is used to hold all the cars in the current board in a Car array list

obstacle : ArrayList<obstacles>: This attribute is used to hold all the obstacles in the current board in an Obstacle array list

height : int : this attribute is used to identify the board' s heigh based on the game mode. If it is the single mode game the height is 6 if it is the multiplayer game it is 22.

width : int: this attribute is used to identify the board's width based on the game mode. If it is the single mode game the width will be 6 but if it is the multiplayer game the width is 14.

win : boolean: the attributes used for the winning situation of the player based on the player's cars position.

Constructor:

Board(gameMode : boolean): In this constructor, initial board state is set (the attributes shown above) by the information passed down from the parents and the game mode parameter show the mode of the game so that board can initialize its size based on the mode. Based on the attributes, we set the validity of the grids.

Methods:

moveCar(i : int, j : int, iDragged : int, jDragged : int, turn : boolean) : boolean: this method used for moving the cars in the board. We consider the drag of the user. Until user came across with invalid grid it can move. Also, the turn is used for the multiplayer's mode. If it is the single player mode the turn is always true for the user but when it is the multiplayer mode the turn s based on the movements of the two players. It returns whether it moved or not

moveBoard(i : int, j : int, iDragged : int) : boolean: this method used for moving the board in the screen. The board can drag only horizontally so that we get only iDragged not jDragged as a parameter. It returns whether it moved or not

setCar(i1 : int, i2 : int, j1 : int, j2 : int, carType : int): This method is used to create a car with the specified horizontal or vertical direction and size, and to place that car to the coordinates given. Every car has a value that represents a car belongs to player 1's or player 2's specific cars or a normal car.

setObstacle(i : int, j : int) This method is used to create an obstacle and to place that obstacle to the coordinates given. There is no need to specify direction or size for an obstacle because all obstacles will be 1x1 size and won't be intractable.

searchCoordinates(i : int, j : int) : Car This method is used to ask all the cars in the current board about their coordinate, if a car confirms that its coordinates are the same with the ones given in the parameter, then this method will return information about the validity of that coordinate. This method returns the car in that coordinates.

checkWin() : boolean: This method used for checking the current situation that any of the players have win or game still continuous. If still continuous it will return false, otherwise, it will return true.

4.3.9. Car

Attributes:

direction : boolean : This attribute is used to display the direction of the car either horizontally or vertically

carType : int : This attribute is used to differentiate the user's car (which needs to be freed in order to win) from the other cars. Because this game has multiplayer, we define player's car as 1 & 2 (for the two players) and a normal car as 0.

size : int : This attribute is used to display the 1-dimensional size of the car since all the cars will be of size 1xN or Nx1

i1 : int : This attribute is the first x coordinate(beginning of the car) of the car we represent it as i1 because of searching in the array with the x y values cause errors because of misunderstanding.

i2 : int : This attribute is the second x coordinate(end of the car) of the car

j1 : int : This attribute is the first y coordinate(beginning of the car) of the car

j2 : int : This attribute is the second y coordinate(end of the car) of the car

Constructor :

Car(i1 : int, i2 : int, j1 : int, j2 : int, carType : int) : This constructor sets the initial attributes of this car to the given parameters.

Methods:

searchCar(i : int, j : int) : boolean : It searches given point whether this car occupies that location or not and returns boolean.

setCoordinates(iDragged : int, jDragged : int) : This method sets the coordinates of the car with the given dragged parameter.

setCarType(carType : int) : This methods sets the car type of the Car with given parameter. We have 3 different number that represents the player's car as 1 & 2 (for the two players) and a normal car as 0.

elevate(i : int) : This method is used to elevate the car if we move the board. Therefore it changes its y coordinate(i1, i2) based on the parameter

getCarType() : int : It returns the car type of the car

4.3.10. Obstacle:

Attributes:

i : int : This attribute is the y coordinate of the obstacle. We represent it as i because we are using arrays to represent the board, i of the array shows the y coordinate.

j : int : This attribute is the x coordinate of the obstacle. We represent it as j because we are using arrays to represent the board, j of the array shows the x coordinate.

Constructors:

Obstacle(i : int, j : int) : This constructor is to initialize obstacle in a specific place.

Methods:

elevate(i : int) : This method is used to elevate the obstacles if we move the board. Therefore, it changes its y coordinate(i) based on the parameter.

4.3.11. Card:

Attributes:

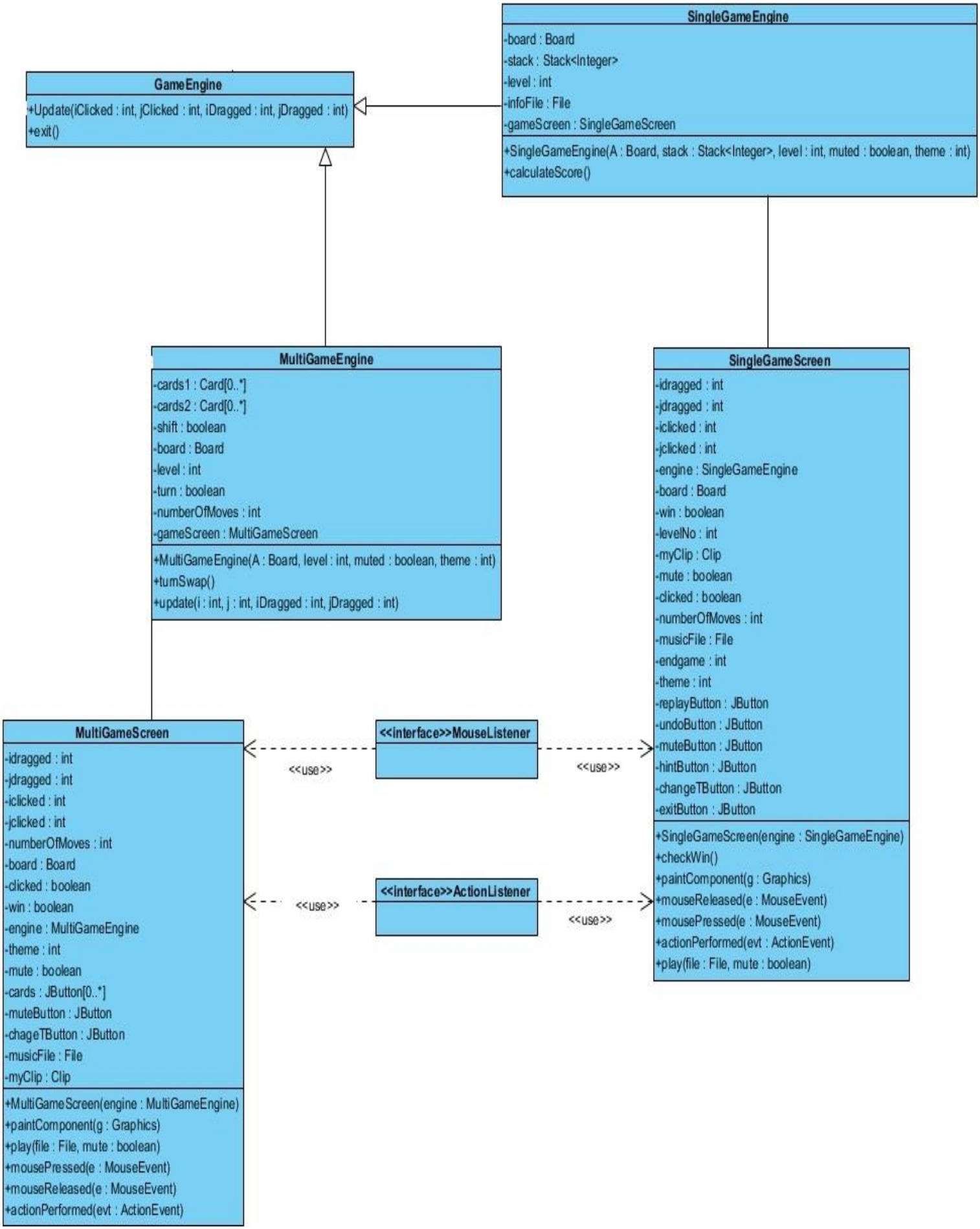
numberOfMoves : int : this attribute shows the number of moves that a card could have. One card can have 1, 2, 3, 4 moves or it can have unlimited moves with one car and only one direction.

shift : boolean : this show that if the card has shift board property or not.

Constructor:

Card() : this constructor initialize random cards. That can be the only number of moves (only 1,2,3,4) or moves and shift (1+shift, 2+shift, 3+shift, 4+shift) or only shift card, or only 1x card which represent the unlimited move in one direction for one car.

Piece 3, Game Management & In Game Screens:



4.3.12. GameEngine Class

GameEngine is an abstract class that only has abstract update method. There are 2 kind of game engine SingleGameEngine and MultiGameEngine.

Methods:

update(i : int, j : int, iDragged : int, jDragged : int): Abstract method. It will be override according to single game or multiplayer game.

4.3.13. SingleGameEngine Class

SingleGameEngine class is the manager class of single game mode. It is responsible from data management of the game. It gets initial data: initial board, highest scores at that level and update these data according to player's actions. Each player moves updates the board and high scores in update method from super class GameEngine.

Attributes:

board : Board: Board is the initialized game board comes from level selection. The board is updated by update method.

stack : Stack<Integer>: Stack is keept the i1,i2,j1,j2 coordinates for the objects due to supplying undo feature. Each undo button pressed, the 4 coordinates are popped so after each player move new 4 coordinates are pushed this stack.

level : int : This integer represents which level are displayed at that time. It comes from DisplayLevelScreen class.

infoFile : File: System keeps a file that stores highest scores for each level and the last level user played. It will be useful to demonstrate highest scores for each level and also resume the game if players want to continue the game from place they leave.

gameScreen : SingleGameScreen: This is the gameScreen of the game. It supplies the system to communication between data management and screens. Each data updates are reflected on the screen with gameScreen.

Constructor:

SingleGameEngine() : It gets data and calls SingleGameScreen gameScreen.

Methods:

calculateScore(): It calculates the score according to number of movements and time.

4.3.14. MultiGameEngine Class

MultiGameEngine class is data controller of the multiplayer game. It gets the initial board from DisplayLevelScreen class. It allows to player move cars and board according to card they played.

Attributes:

cards1 : Card [0…*]: Cards 1 keeps the card rooster of player 1.

cards2 : Card [0…*]: Cards 2 keeps the card rooster of player 2.

board : Board: As being SingleGameEngine class, the board is initialized in DisplayLevelScreen class and comes to MultiGameEngine class. The board is changed by update method according to players' decisions.

level : int: This integer demonstrates which board configuration are displayed at that time.

numberOfMoves : int: Players always have 4 cards and can play with them. Cards have numberOfMovement data. After player plays a card numberOfMoves gets from that card.

turn : boolean: Boolean turn shows which user is playing at that moment. It is used to avoid that one player touch the other player's car.

shift : boolean: Boolean shifts checks whether player decide and move board according to card content.

gameScreen : MultiGameScreen: This is the gameScreen of the game. Updated Data are sent to gameScreen and gameScreen paint it on screen. After each move gameScreen updates data thanks to MultiGameEngine class.

Constructor:

MultiGameEngine(): It gets data, initialize cards and calls MultiGameScreen gameScreen.

Methods:

turnSwap(): Turn swap method changes the turn of players and draw a new card for them. The new card is added player's rooster.

update(i : int, j : int, iDragged : int, jDragged : int): Update method changes positions of cars or board according to i, j, iDragged, jDragged data. These data come from listeners of gameScreen.

4.3.15. SingleGameScreen Class

SingleGameScreen class is the painter of single game. It communicates with SingleGameEngine engine. It gets the actions by using button and mouse listeners. It sends data to engine. Engine updates data and send back to this class gameScreen. GameScreen gets updated data and paint it again and again. GameScreen checks whether the player wins or not. It allows players to turn back to last movement (undo), exiting game, turn back to main menu, mute music, change theme, get hint, replay.

Attributes:

iPressed : int: it is the horizontal instances of point that user press the object.

jPressed : int: it is the vertical instances of point that user press the object.

iReleased : int: it is the horizontal instances of point that user releases the object.

jReleased : int: it is the vertical instances of point that user releases the object.

levelNo : int: it is indicator of which level are played at that moment. If player cannot win the game, this data is stored in file. After player runs the game, he/she can resume the game at this level.

endgame : int: It is the output of JOptionPane. User might go to main menu, next level or exit the game according to this choice.

theme : int: It represents which theme are active at that moment.

numberOfMoves : int: It increases each successful movements. It is important to determine score.

engine : SingleGameEngine: Engine is data manager of the game. Therefore, engine update boards according to player's actions.

board : Board: Game board comes from the game engine. Engine updates board and board is painted in here.

win : boolean: It shows whether the player wins the game.

mute : boolean: It shows the frame is mute or not. It is changed by mute button.

clicked : boolean: It solves a bug that we've encountered in mouse releasing. Basically, it makes the releasement happen only once.

myClip : Clip: Clip is a particular java class plays the wav. music files.

musicFile : File: It is a wav file to play music on frame.

muteButton : JButton: If music is active, it mutes the frame. If frame is muted, it starts the music again.

changeTButton : JButton: It changes the theme.

exitButton : JButton: It allows user to exit game or turn back to main menu.

replayButton : JButton: It starts the game again.

undoButton : JButton: It goes 1 movement back. Until we reach the point where the stack is the initial stack, we can do undo, what different about hint is that we can use hint after that point too.

hintButton : JButton: It gets one hint(movement) to user. It is kept in stack already. Stack consists of 2 parts, one is from the case where the red car is at the exit line, to the initial board, and the other part is from that initial board to the current board. The number of elements of the first part will be known since we know which level, we're in.

Constructor:

SingleGameScreen(engine : SingleGameEngine) : Constructor of SingleGameScreen initialize the frames width, heights, buttons and their positions, plays music initially.

Methods:

checkWin(): It checks whether player's car reach the exit.

paintComponent(g : Graphics): Paint component is the original java class that paint all graphs, images, texts. All GUI parts is painted in this method.

mouseRelased(e : MouseEvent): mouseReleased method gets the x, y coordinates when mouse pressed and turning into node coordinates by division of the frame width and height. After that it calls engine to update board according to new positions of cars and obstacles.

mousePressed(e : MouseEvent): mousePressed method gets the x, y coordinates when mouse released and turning into node coordinates by division of the frame width and height.

play(file : File, mute : boolean): It gets musicfile(.wav) and play it by checking boolean mute.

actionPerformed(e : ActionEvent): Action performed controls the buttons. If players pressed any buttons, this method handles appropriate changes.

4.3.16. MultiGameScreen Class

MultiGameScreen class is the painter of multiplayer game. It communicates with MultiGameEngine engine. It gets the actions by using button and mouse listeners. Game engine gets new data thanks to listeners of this class. After Game engine updates data, game screen paints new state according to new data. Game screen checks whether the player1 or player2 win. It also allows players to exiting game, change theme, mute music. In multiplayer game cards is demonstrated as buttons and images. Players manage their actions by pressing card buttons.

Attributes:

iPressed : int: it is the horizontal instances of point that user press the object.

jPressed : int: it is the vertical instances of point that user press the object.

iReleased : int: it is the horizontal instances of point that user releases the object.

jReleased : int: it is the vertical instances of point that user releases the object.

endgame : int: It is the output of JOptionPane. User might go to main menu, next level or exit the game according to this choice.

theme : int: It represents which theme are active at that moment.

numberOfMoves : int: It increases each successful movement. It is important to determine score.

engine : MultiGameEngine: Engine is data manager of the game. Therefore, engine update boards according to player's actions.

board : Board: Game board comes from the game engine. Engine updates board and board is painted in here. Multi game board also might be shifted by players in this game mode.

win : boolean: It shows whether the player wins the game.

mute : boolean: It shows the frame is mute or not. It is changed by mute button.

clicked : boolean: It solves a bug that we've encountered in mouse releasing. Basically, it makes the releasement happen only once.

myClip : Clip: Clip is a particular java class plays the wav. music files.

musicFile : File: It is a wav file to play music on frame.

muteButton : JButton: If music is active, it mutes the frame. If frame is muted, it starts the music again.

cards : JButton[0…*]: Cards of players are represented as JButtons in multi game screen. Players pressed the buttons to play cards.

changeTButton : JButton: It changes the theme.

exitButton : JButton: It allows user to exit game or turn back to main menu.

Constructor:

MultiGameScreen(engine : MultiGameEngine): Constructor of MultiGameScreen initialize the frames width, heights, buttons and their positions, plays music initially.

Methods:

paintComponent(g : Graphics): Paint component is the original java class that paint all graphs, images, texts. All GUI parts is painted in this method.

mouseRelased(e : MouseEvent): mouseReleased method gets the x, y coordinates when mouse pressed and turning into node coordinates by division of the frame width and height. After that it calls engine to update board according to new positions of board, cars and obstacles.

mousePressed(e : MouseEvent): mousePressed method gets the x, y coordinates when mouse released and turning into node coordinates by division of the frame width and height.

play(file : File, mute : boolean): It gets musicfile(.wav) and play it by checking boolean mute.

actionPerformed(e : ActionEvent): Action performed controls the buttons. If players pressed any buttons, this method handles appropriate changes.

4.4. Deployment Diagram

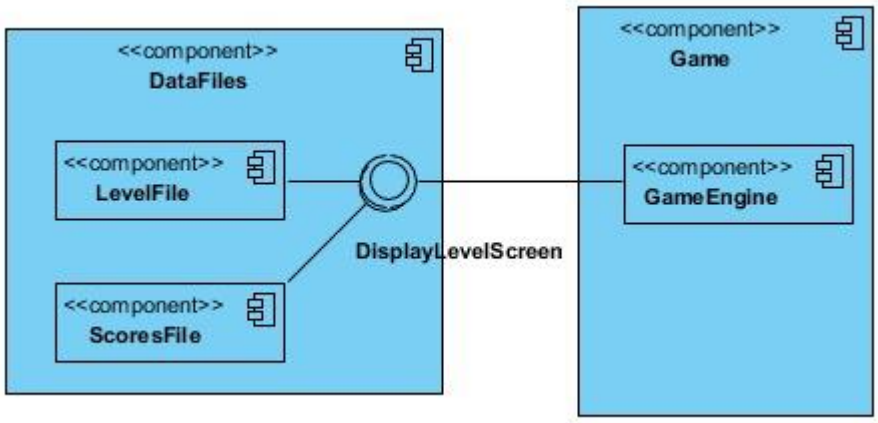


Diagram that shows the intreaction between the info file and display level screen class. Our levels' information is in the LevelFile and we give this information through the DisplayLevelClass.Also we have the ScoresFile which keeps the highest scores of the user and the paused level information and DisplayLevelSelection class uses and updates that information.

5. Improvement Summary

Since the last iteration of design and analysis reports are sent, we have made important changes to our game. We will conclude the changes in steps below. With the significant changes we have made, our design is also changed in particular ways in time. We have clear trade-offs for the low-level design and accordingly high-level design also.

- The first and most important change is the platform which we implemented our game with. Instead of using Unity with coding in C#, we have implemented our game in Java. The most impact on the implementation is changing the system decomposition from the beginning and having different user interface also. Now, we have MVC design in our system decomposition and we have three main decompositions for UserInterface, Model and Game Management.

- The second big change in the game is a Multi-player option for play. Now, the multiplayer section is more complicated than the first design of the game. It is connected to the basic features of the Single Player and also have its own special features as mentioned above chapters.
- We have added extra classes in both user interface and additional functionalities such as Portal and Bridge. For UI, we basically have now 2 main screens: Main Screen and In-game Screen.
- We have added new features. One is Portal and the other is Bridge. The idea of these features belongs to us founded later on Field Visit. They both add more fun to the game and also deepness for the strategy of the game. But their implementations are not yet over.
- The decisions over the database of the game have been made. To be simpler and useful for our game, we have implemented a file-based storage for the scores of the player and the level that the player last entered. As we do not much include complexity to the data side of the game, we thought that would be better for memory wise.
- Packages that include the items of the game are updated as the themes and the game is extended in the developer parts. Also, as we transferred the implementation of the game to the Java, now we have packages from Java supports.
- Object design is changed as we added more items and connections between them changed. The functionalities and the calculations on the game engine are changed. So, the class structure of the management changed with the object design.
- Relation between engine and other classes is updated.
- Car type is set to distinguish the red car from others.

6. Contributions in Second Iteration

All parts revised by M. Said Demir, Ata Coşkun, Zeynep Nur Öztürk, Asuman Aydın, Tarık Emin Kaplan.