CS 319 - Object-Oriented Software Engineering

System Design Report

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 strategy solving the 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. Also, it has specific qualities from distracting from the original game. For instance, one of them is obstacles where pushes to the player to think more and have more complex and nice appearance. Player’s goal is to take the specific car out from the parking lot where there are other cars in different shapes and sizes. 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.

1.2.1 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 that not tiring the eyes of a player. 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. Since we need UI support, we generally get help from the Unity GUI features (see 4.3) but most of all the design and implementation of UI is again up to us.

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 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. Because there is the help of the Unity features (see 4.3), we used the support of reusability of such features.

Portability:

The player uses a mobile phone maybe but they may want to change the platform where they play the game. For this purpose, the intention is to use the help of Unity (see 4.3). There is Multiplatform Support in Unity that allows the developer to build once and then transfer it to platforms such as IOS and Android etc.

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:

User Interface (UI)

Graphic User Interface (GUI)

2. System Architecture

2.1 Subsystem Decomposition

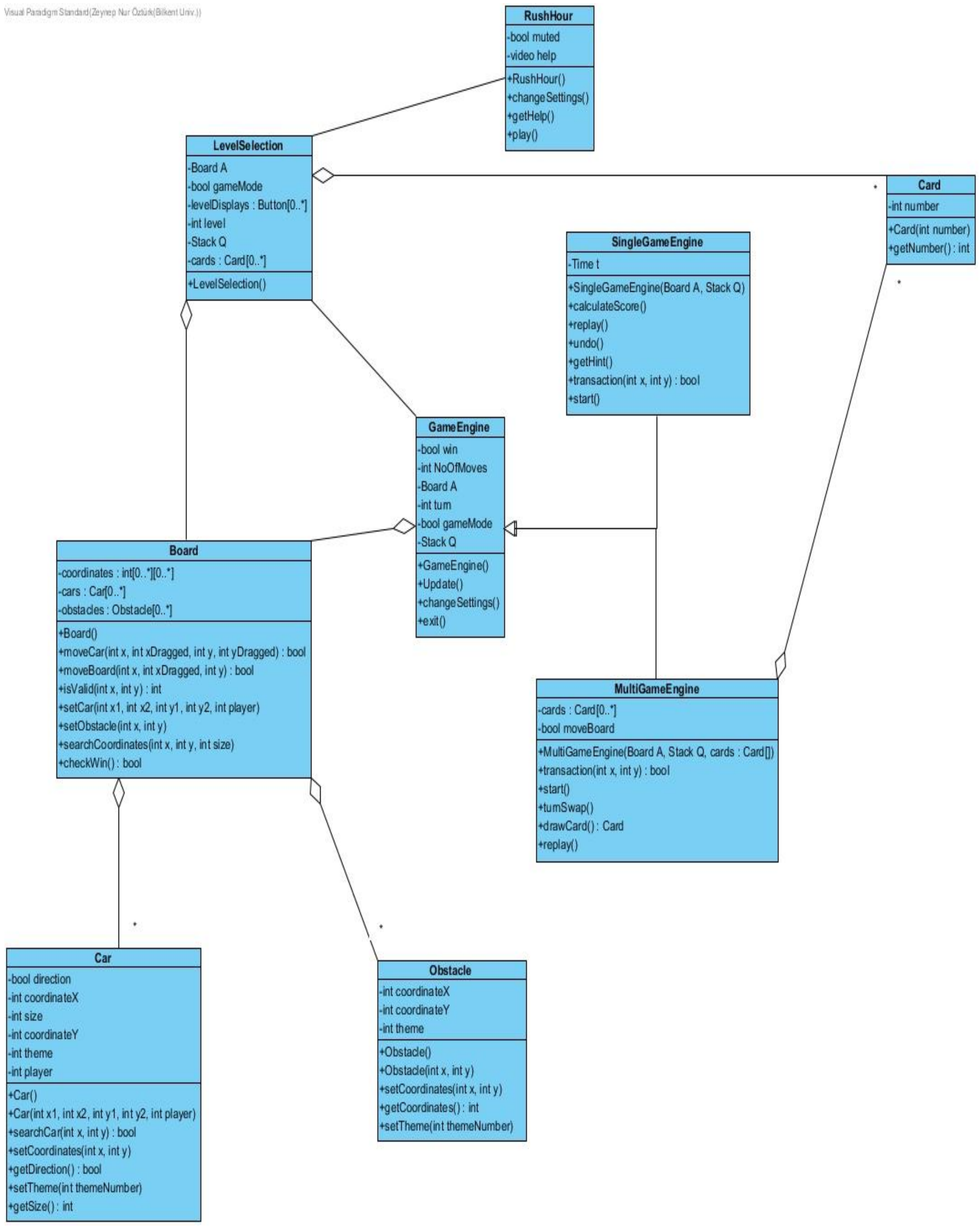
We disintegrate our system into subsystems. Our purpose is to enlarge the compatibility of the components, decrease the coupling of separate subsystems. Modifying or updating the game will be easier with the decomposition of our game.

We have decided that Unity’s GUI libraries will be great to our game’s structure. We disintegrate our system based on unity’s GUI library. All of the codes will be written in Unity but we won’t use any of unity’s library except the GUI part. We will be coding all of the classes in our class diagram by hand. (see 4.3)

As we indicate above, we divide our system into three subsystems. User Interface, Game Engine, and Game Objects are our three subsystems. These three subsystems have different features. Differently, every system can invoke other systems to have a sustainable game. User Interface can apply a functional action to Game Objects to create a board or Game Engine to starts the game. Hence, it allows us to solve problems, bugs, and errors easier and makes the game stable, modifiable, and expandable.

In the Game Engine subsystem, we control all of the game. It also controls the GUI part. Update method checking every single move and updating the GUI. It has two inheritance class in it to maintain both the single player and the multiplayer because two of has different implementation and have some common features. Firstly, Board class will create a beginning map of our system and it will be connected to the Game Engine Class and will be updated continuously. And every update will check the wining circumstance. If the player wins, the score will be calculated or it will keep update itself until the user stops or replay it.

As a conclusion, we decompose our game to have high cohesion and less coupling. Our system will make our game more expandable and flexible, maintainable.



2.2 Hardware/Software Mapping

Rush Hour game will be implemented in Unity and by using C# for script language. Our first priority is to make this a mobile game however since we are using unity and it's a language with high portability it can be played on PC as well. For running the game, the requirements are the same with running any other Unity game [3].

Software requirements:

Valid Operating Systems for PC: Windows 7 SP1 (or higher), or macOS 10.11 (or higher), or Ubuntu 12.04 (or higher), or SteamOS

iOS player requires iOS 8.0 or higher.

Android: OS 4.1 (or higher) and OpenGL ES 2.0 (or higher).

For hardware requirements, player needs to have an Android phone (having either ARMv7 CPU with NEON support or Atom CPU and a functioning touch screen) or an iOS phone (with functioning touch screen) satisfying the software requirements specified above, or for PC, the hardware needs are:

- Graphics card with DX10 (shader model 4.0) capabilities.
- CPU: SSE2 instruction set support.
- Any functioning mouse

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 touch screen of their phones (or with their mouse, for PC)

2.3 Persistent Data Management

There is no need for any sort of complex database interaction because 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, 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. Furthermore, if we would have enough time, we're thinking about adding a resume button to the main menu which will make you able to continue to the last game you've played. For this implementation, we will store the required data (about the last entered game) to a text file as well. And even furthermore, if we would have extra-extra time left, we will implement a multiplayer game which 2 players race against each other from a network cloud. For this implementation, we will upload the boards of the players to cloud and players will see each other's boards and which state they are in. For this implementation, we will store the current boards to cloud in a text file. But again, these 2 features are not considered within the current system.

2.4 Access Control and Security

There is no need for any security and access control measures, considering there won't be any network connection required operations. 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 hardware.

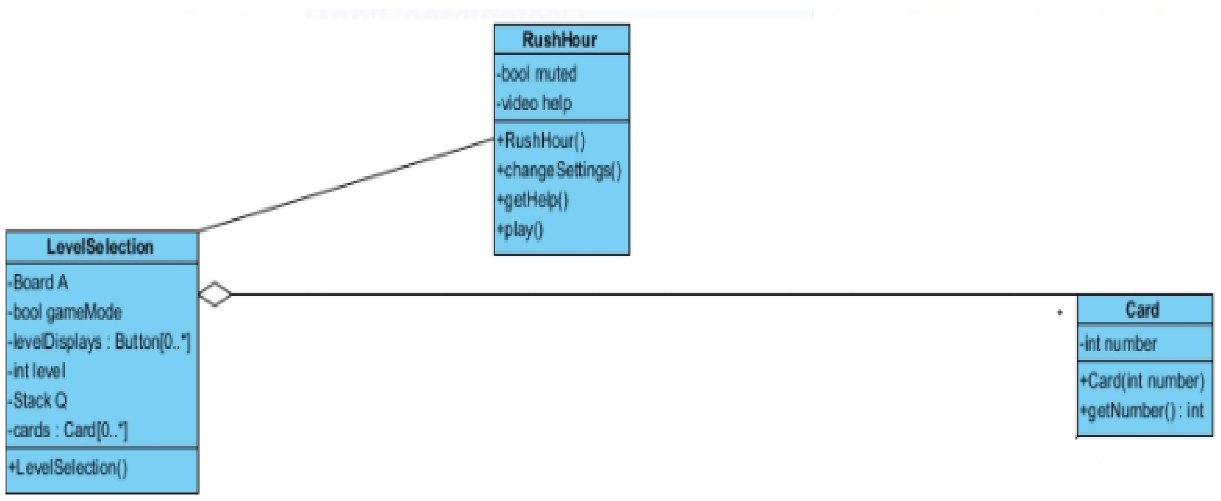
2.5 Boundary Conditions

Rush hour needs to be installed only once on your hardware in order to play it, there will be an executable file. Since we intend to make this executable 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 back button (or its equivalent, if the phone doesn't have that) on phone in the main menu, or by pressing the "X" on the top right of the window on PC.

If for any reason the game crashes, the player's current progression, scores and personalized settings won't 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).

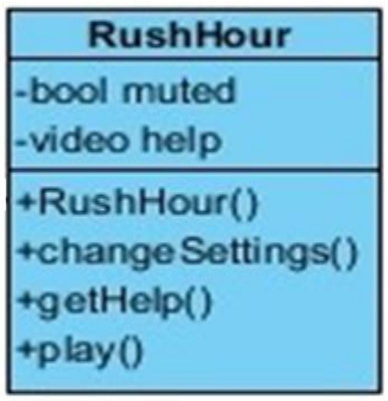
3. Subsystem Services

3.1. User Interface Subsystem



User Interface Subsystem consists of three classes and is responsible for the establishment of the interface between the system and the user. The RushHour class will be the first thing that the user will see when the user entered the system. It will consist of several buttons (as we discussed in the previous report by Mock-Ups) that will represent the functions of the RushHour class. Settings will display the changeSettings function which will include the options such as mute sound or theme. Help button will display the tutorial video and play will redirect the user to the LevelSelection class. At this stage, player will see two options which will represent the gameMode options such as single or multi player, then user will see the level menu where the user can choose from listed levels each includes basic image of description of the level and high score and the difficulty of that level. As for the cards, the user will be able to see them at the multiplayer game mode as we will discuss further at upcoming parts.

3.1.1. RushHour Class



ATTRIBUTES:

- bool muted: it is to check whether the sound is on or off. It will be true If the sound is on and false when the sound is off.
- Video help: It includes a tutorial that explains the game and how to play it.

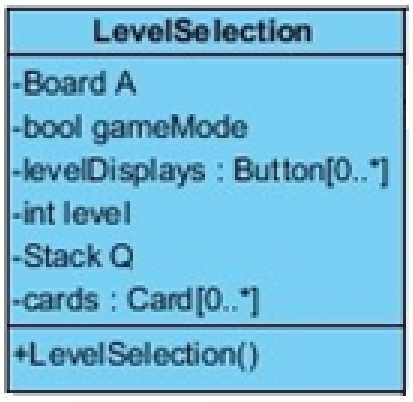
CONSTRUCTOR:

- RushHour(): It is a constructor that initialize the given attributes and make calls for starting the game. The main menu is started with this class.

METHODS:

- changeSettings(): Settings are changing the sound and changing the theme of the game. It has if conditions which include bool muted checked. Also, it includes the theme info as an integer list and it displays the flow of themes when the user chooses to change the theme. To change the theme, it makes a call of methods from children classes.
- getHelp(): it basically shows the help video.
- Play(): This method makes the call for level selection class' constructor to initialize the game.

3.1.2. LevelSelection Class



This class has a "has" relationship with Card and Board class.

ATTRIBUTES:

Board A: It is an instance of Board class to set the board game when the user chooses the player mode and level of the game if it is single mode.

Bool gameMode: It is 0 if the game mode is single and it is 1 if the game mode is multi.

levelDisplays: Button[0..*]: The buttons belong to the levels in the screen. For instance, the levels are associated with the button array elements and a level is chosen it will be connected to the corresponding button array element.

int level: This is the number of specific level to associate with the button array.

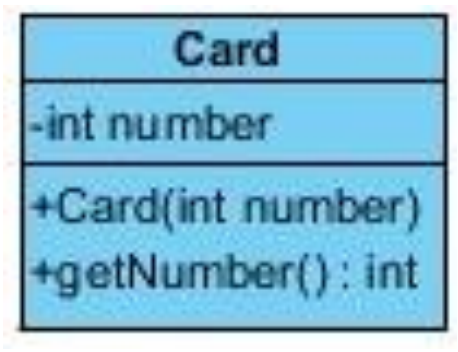
Stack Q: The stack is used here to keep the moves of the player with storing board list in it. So, in the future specifications, we can give the hint. The levels' solution information is pushed into stack first. For example, the easiest solution requires 6 moves, so 6 different boards will be stored in this stack. The 6th board will be initial board and the previous ones will be the steps that is required to win. And when the user moves the cars, it again pushes the current board to the stack Q. When the user wants a hint, one by one, the moves are popped out of stack Q and from solution part, one move is displayed. (it will be further discussed)

Cards: Card[0..*]: The cards part of the implementation of this class belongs to the multiplayer mode. In multiplayer mode, there are cards to draw by each player. They play the game according to the specifications in the cards. This array is corresponding the specific cards defined by the cards class. (it will be further discussed)

CONSTRUCTOR:

LevelSelection(): As in RushHour class, play function makes a call of level selection constructor, it is the usage for starting the game. Hence, this constructor starts the game and waits for the user to choose the player mode. If it is a single player mode, it shows the user the levels. When the user chooses one level, the button corresponding to that is taken into consideration. For each button array element, there will be if statement and when one of them is chosen, Board A will be set with its properties and SingleGameEngine will be called to start the game for real. If it is a multiplayer mode, there will be few levels to choose as there are cards that add more feature the game. From the levels, as in the single-player mode, the level will be chosen and according to the if statement for that level will be executed with it is statements in it such as MultiGameEngine constructor call.

3.1.3. Card Class



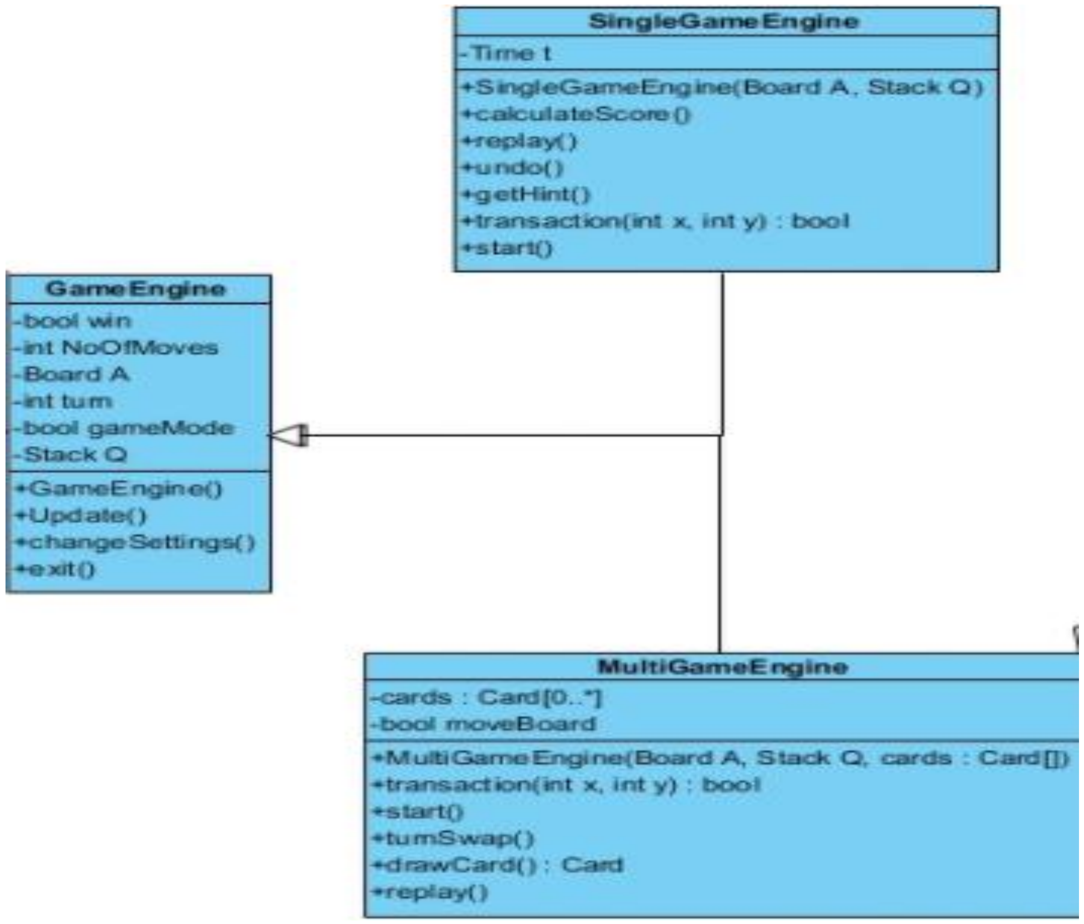
ATTRIBUTES:

- int number: it keeps the number which the card has a move count.

CONSTRUCTOR:

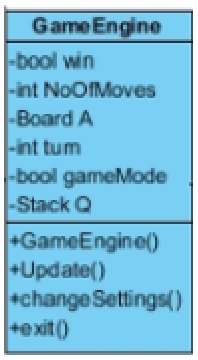
- Card(int number): It is a constructor to set the number of the card.
- getNumber(): int: It is a helper function to allow parent classes to access the number information.

3.2 Game Management Subsystem



Game Management Subsystem is consisting of three classes which are two subclasses inherited to the another one. Since SingleGameEngine and MultiGameEngine both needs to use some same attributes and methods, we find it logical to make them the subclasses of the GameEngine class. They are the main controller classes for their own type of games.

3.2.1. GameEngine Class



Game Engine is the fundamental class of rush hour game. All actions about the game such as moves of cars, drawing of cards, calculation of scores, time limitations, board moves, options that changes themes and sound, keep and change turns if it is multiplayer mode, undo, getting hint and also communication with device such as conversion of the board grid layout according to percentage of pixel configuration, calculation of which point is clicked and whether that point is in board, what the point is on grid representation of board is handled in this class. GameEngine class will have 2 sub classes SingleGameEngine class and MultiGameEngine class.

ATTRIBUTES:

Board A: This board comes from LevelSelection as initialized. This board will be the game board and all game actions is handled on that board.

int NoOfMoves: This represents number of moves in the game. If user plays in single mode, number of moves indicate how many stars player get at the end of the game. If player exceeds the normal (depends of difficulty) number of moves, he/she will get less stars. Therefore, after each move, NoOfMoves will increase. However, if users play in multiplayer mode, NoOfMoves come from the drawn card. The cards might have number of movements cars and obstacles or board. If the NoOfMoves reach the number of movements on card player turns ended and swap. Therefore, NoOfMoves will decrease in the Multiplayer mode.

int turn: This represents the which player is playing at that moment. This variable is mostly important for multiplayer mode because it does not change already in the single player mode. This variable will be 1 for player1 and 2 for player2. It is managed by turnSwap method in MultiGameEngine subclass.

bool win: This variable indicates whether the game is over. The infinite game loop will check this variable is 0 or 1 to ended the game or not.

bool GameMode: This boolean demonstrates that which game mode single or multiplayer is selected by user. After that GameEngine call the SingleGameEngine or MultiGameEngine.

Stack Q: Q stack will keep the initial solution of the game. For example, if solution of one level is 7 move this stack will keep the initial 7 board situation. This board states pushed in LevelSelection to stack. If the player want to get hint, engine gets the board states from this stack. In addition, all moves pushed into stack because if players want to click undo, the last board state popped from this stack.

CONSTRUCTOR:

GameEngine() : When GameEngine is called it takes the boolean GameMode and call SingleGameEngine or MultiGameEngine subclasses.

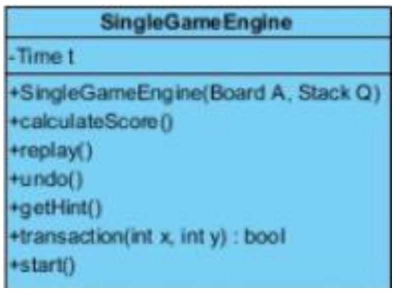
METHODS:

Update(): All GUI parts and components is adjusted and handled in update method. After all changes is managed on Board A update method paint and create GUI components according to Board A.

changeSettings(): This method will adjust theme or sound according to which button is clicked by user.

exit(): If player exit the game manually, system exit the infinite game loop and the game ends.

3.2.2. SingleGameEngine Class



SingleGameEngine class is a subclass of GameEngine class. This class is customized for single player game because single player game differentiates from multiplayer with some features such has difficulty levels, time limitations, getting stars, scores, undo and hint options. The infinite game loop is settled in this class. Therefore, actions will start in this class and continue until game is over.

ATTRIBUTES:

Time t: If there is time challenge in the game, it determines the time.

CONSTRUCTOR:

SingleGameEngine(Board A, Stack Q) : Constructor adjust and gives initial conditions to game loop.

METHODS:

start(): Method will start infinite game loop.

calculateScore(): This method will calculate score according to number of movements and time.

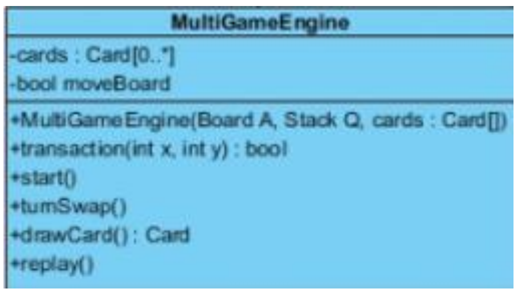
replay(): The game will start with initial conditions again.

undo(): Undo method pop the last state of board from Q stack if the stack is not in initial condition. Therefore, player cannot call undo if he/she does not any move until that time. User can use undo just only until initial condition.

getHint(): If player wants to get hint, this method pops the board states from Q stack until initial condition first and then continue to show solution steps to player.

transaction(int x, int y): The method is one of the key methods for communicating with device. When users tap the screen, it gets the x, y pixel coordinates first. After that determine whether these x, y is in the board or not. If the point is in board, it indicates which grid node the point is in. After that game engine can manage the changes in itself according to this transaction.

3.2.3 MultiGameEngine Class



MultiGame class is one of the subclasses of GameEngine class. This class is created for multiplayer game mode. Multiplayer game mode's nature is different than single player mode.

In this mode, players will draw a card in each turn. Therefore, the game will depend on a little bit of luck. Moreover, players can move the board according to card contents. The game loop is settled in class and with start command game starts. If any player wins the game, the game ends. When the number of movements of a player ends, the turn will be passed on to the other player

ATTRIBUTES:

Card [] cards: Card array includes game cards.

bool moveBoard: It check whether the card include move board feature

CONSTRUCTOR:

MultiGameEngine(Board A, Stack Q, Cards[]) : Constructor adjust and gives initial conditions to game loop.

METHODS:

start(): Method will start infinite game loop.

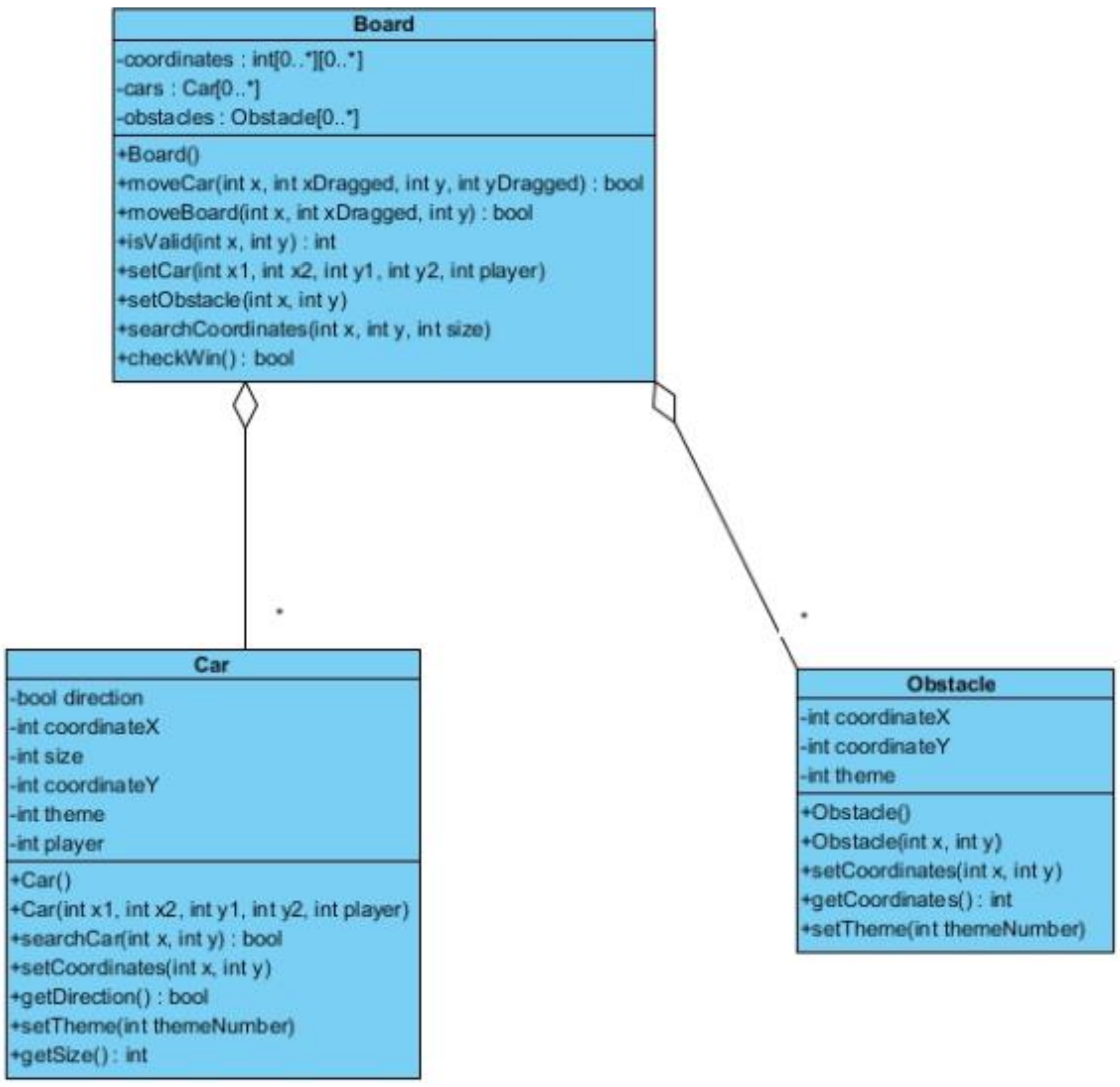
transaction(int x, int y): The method is one of the key methods for communicating with device. When users tap the screen, it gets the x, y pixel coordinates first. After that determine whether these x, y is in the board or not. If the point is in board, it indicates which grid node the point is in. After that game engine can manage the changes in itself according to this transaction.

replay(): The game will start with initial conditions again.

drawCard():   Each turn players will draw a card. The card is taken from the card array which is initialized in advance in LevelSelection class and return this card.

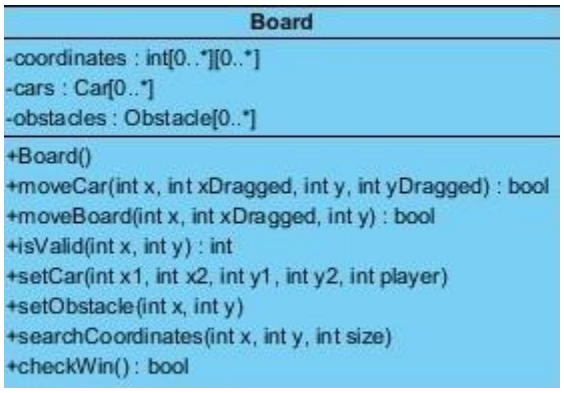
turnSwap: In multiplayer game we have more than 1 players so we have to indicate which player is moving at that moment. We should not forget that active player cannot change the passive player’s car. Cars will have a number which determines the kind of cars (it will be discussed further). turnSwap also important because of that. After each move, turnSwap update the int turn variable.

3.3 Game Objects Subsystem


Board
-coordinates : int[0..*][0..*]
-cars : Car[0..*]
-obstacles : Obstacle[0..*]
+Board()
+moveCar(int x, int xDragged, int y, int yDragged) : bool
+moveBoard(int x, int xDragged, int y) : bool
+isValid(int x, int y) : int
+setCar(int x1, int x2, int y1, int y2, int player)
+setObstacle(int x, int y)
+searchCoordinates(int x, int y, int size)
+checkWin() : bool
*
*
Car
-bool direction
-int coordinateX
-int size
-int coordinateY
-int theme
-int player
+Car()
+Car(int x1, int x2, int y1, int y2, int player)
+searchCar(int x, int y) : bool
+setCoordinates(int x, int y)
+getDirection() : bool
+setTheme(int themeNumber)
+getSize() : int
Obstacle
-int coordinateX
-int coordinateY
-int theme
+Obstacle()
+Obstacle(int x, int y)
+setCoordinates(int x, int y)
+getCoordinates() : int
+setTheme(int themeNumber)

Game Objects Subsystem consists of three classes which stores the value about the game objects. They will be the classes that the game engine will control and change depending the situation. The main objects of the game are cars and obstacles that is 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 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.

3.3.1 Board Class



```
Board
-coordinates : int[0..*][0..*]
-cars : Car[0..*]
-obstacles : Obstacle[0..*]
+Board()
+moveCar(int x, int xDragged, int y, int yDragged) : bool
+moveBoard(int x, int xDragged, int y) : bool
+isValid(int x, int y) : int
+setCar(int x1, int x2, int y1, int y2, int player)
+setObstacle(int x, int y)
+searchCoordinates(int x, int y, int size)
+checkWin() : bool
```

ATTRIBUTES:

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

Car cars[0..*]: This attribute is used to hold all the cars in the current board in a Car array.

Obstacle obstacles[0..*]: This attribute is used to hold all the obstacles in the current board in an Obstacle array.

CONSTRUCTOR:

Board(): In this constructor, initial board state is set (the attributes shown above) by the information passed down from the parents. Based on those attributes, we we set the validity of the grids.

METHODS:

isValid(int x, int y): This method checks a given point is valid or not. A valid point in grid means that point is not occupied by a car nor obstacle; an invalid point means the point is occupied. This method returns 0 if there nothing in the grid, returns 1 when there is a car in it, 2 when that place belongs to board, returns 3 when it is outside the board.

moveCar(int x, int xDragged, int y, int yDragged) : 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. It returns whether it moved or not

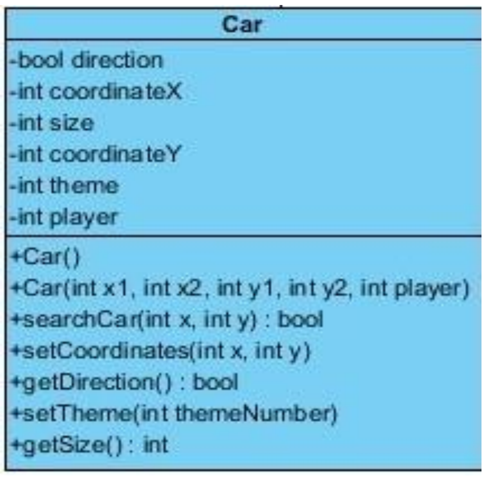
moveBoard(int x, int xDragged, int y) : this method used for moving the board in the screen. The board can be drag only horizontally, so that we get only x dragged not y dragged as parameter. It returns whether it moved or not

setCar(int x1, int x2, int y1, int y2, int player): This method is used to create a car with specified horizontal or vertical direction and size, and to place that car to the coordinates given. Every car has a value that represent a car belongs to player 1 or player 2.

setObstacle(int x, int y): 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 interactable.

searchCoordinates(int x, int y, int size): 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 checkWin() : this method used for checking the current situation that any of the players has win or game still continuous. If still continuous it will return false, otherwise it will return true.

3.3.2 Car Class


Car
-bool direction
-int coordinateX
-int size
-int coordinateY
-int theme
-int player
+Car()
+Car(int x1, int x2, int y1, int y2, int player)
+searchCar(int x, int y) : bool
+setCoordinates(int x, int y)
+getDirection() : bool
+setTheme(int themeNumber)
+getSize() : int

ATTRIBUTES:

int coordinateX: This attribute is the x coordinate of the car

int coordinateY: This attribute is the y coordinate of the car

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

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

int theme: This attribute is the number of themes that is chosen by user int player: 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 multiplayers, we define player’s car as 1 & 2 and a normal car as 0.

CONSTRUCTOR:

Car(): This constructor sets the initial attributes of this car

Car(int x1, int x2, int y1, int y2, int player): This constructor sets the initial attributes of this car to the given parameters.

METHOD:

setCoordinates(int x, int y): This method sets the initial coordinates of the car

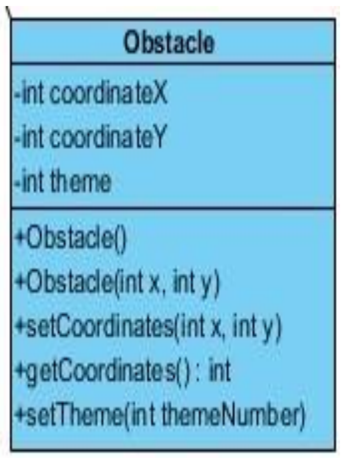
searchCar(int x, int y) : It searches given point whether this car occupies that location or not and returns boolean.

setTheme(int themeNumber): This method is used to set the theme of the car

getDirection(): This method returns the direction of the car as a boolean value to show whether it is horizontal or vertical

getSize(): This method returns the varying size of the car.

3.3.3 Obstacle Class



ATTRIBUTES:

int coordinateX: This attribute is the x coordinate of the obstacle

int coordinateY: This attribute is the y coordinate of the obstacle

int theme: This attribute is the number of themes that is chosen by user

CONSTRUCTOR:

Obstacle(): This constructor sets the initial attributes of this obstacle

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

METHOD:

setCoordinates(int x, int y): This method sets the initial coordinates of the obstacle

getCoordinates(): This method gets the coordinates of the obstacle setTheme(int themeNumber): This method is used to set the theme of the obstacle. Themes are represented by numbers for the information flow inside the game however they correspond to set a different image for an object upon a theme selection

4. Low-level Design

4.1. Object Design Trade-Offs

Understandability vs Functionality:

Every difference we make in the design is aim to have more understandable and not completed implementation. The class diagram of design, now, is very different than the earlier version. It simply makes the job easy to implement with the object-oriented programming logic. When the design is more simplistic to understand better, it is expected that it loses its functionality. However, considering the class and connection between class, functionality also increases. Because, as mentioned earlier, methods and classes have more job to do so that usage of objects increases and separate job distribution decreases.

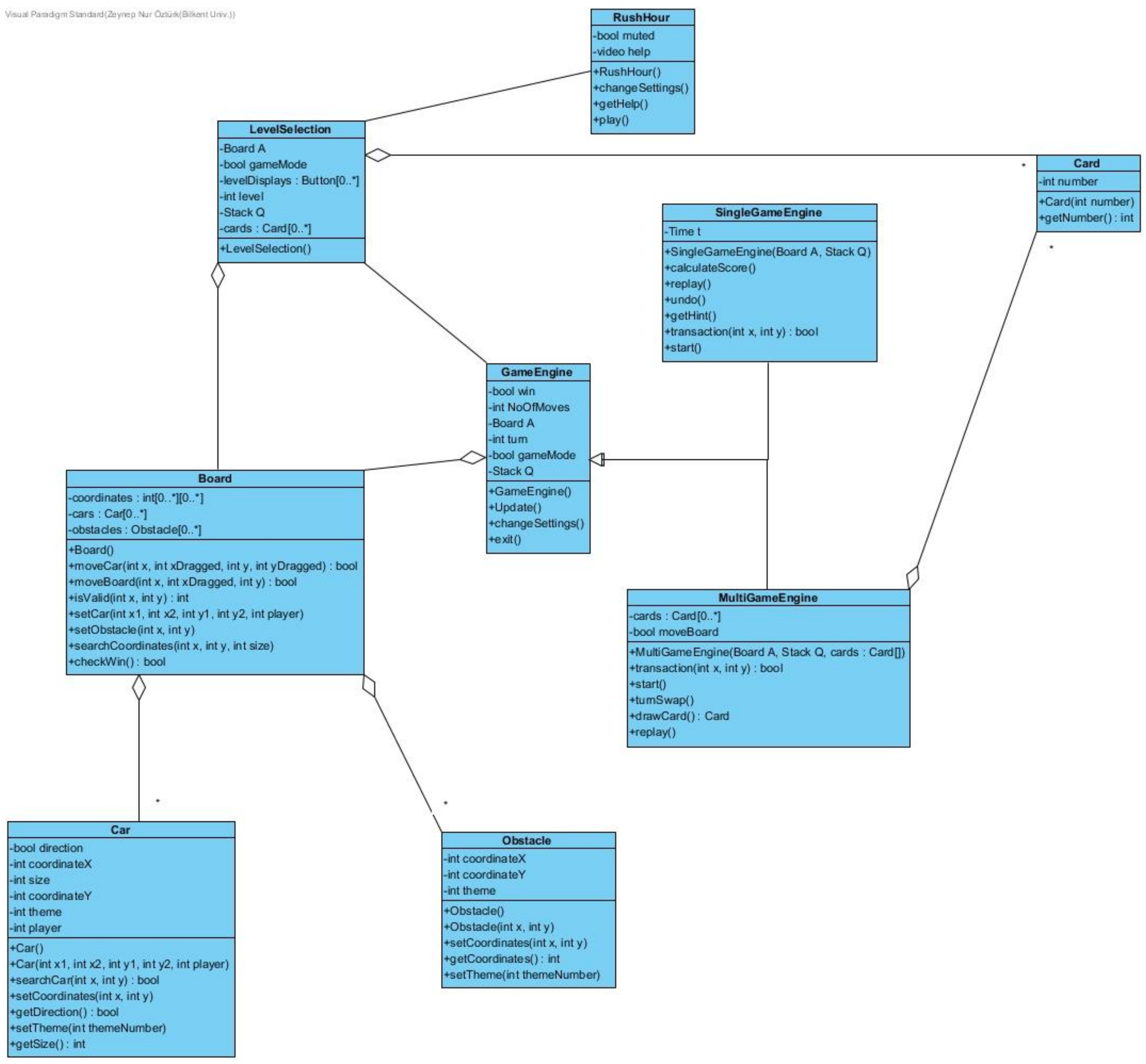
Memory vs Maintainability:

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 that 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. Hence, for this state of design, it appears that memory and maintainability are thought to be well-defined.

Development Time vs User Experience:

For the implementation of UI, we choose to use Unity’s graphics interface support. As Unity has both making the platform transformations easy and has multiple UI class to use. However, we will design all the Board properties and make the changes by the class that we have shown in the class diagram. Hence, by keeping the UI stable with enjoyable way, we also decreased the development time but we also have the functionality to control the game.

4.2. Final object design



4.3. Packages

Some of the features are tended to use in implementation. **But they are not affecting the architecture of game nor design, they are only used to enhance the GUI which we asked to both Eray Hodja and Muhammed Ağabey about whether we can use the GUI enhancer libraries of Unity before.** Also, we will use the Unity’s mobile converter. Unity can convert the written game to a mobile game easily. We’ve asked about this too, whether we’re allowed or not

4.3.1 UnityEngine.IMGUIModule

The GUI class is the interface for Unity's GUI with manual positioning.

4.3.2 UnityEngine.AnimationModule:

AnimationEvent lets you call a script function similar to SendMessage as part of playing back an animation.

Animation events support functions that take zero or one parameter. The parameter can be a float, an int, a string, an object reference, or an AnimationEvent.

4.4. Class Interfaces

4.4.1. Touch

Since this is a mobile game, only interface that we’ll be using is the touch interface. Whenever player clicks on somewhere on the game, this interface will be invoked and will give us the coordination of the location that the player touched. Also, at the moveCar and moveBoard methods, we’ll be using information which is supported by the touch interface.

5. Glossary & References

- Technologies, U. (n.d.). GUI. Retrieved from https://docs.unity3d.com/ScriptReference/GUI.html
- Technologies, U. (n.d.). GUI. Retrieved from https://docs.unity3d.com/ScriptReference/AnimationEvent.html
- System requirements for running Unity games: https://unity3d.com/unity/system-requirements [Accessed: 08.11.2018]