



Bilkent University
Department of Computer Engineering

CS319

Object Oriented Software Engineering

Design Report

OtoParker

Group 7 - Vana Members:

Arda Usman

Ali Çetin

Çelik Köseoğlu

Hüseyin Beyan

1. Introduction	3
1.1.Purpose of the System	3
1.2.Design Goals	3
1.2.1.End User Criteria	3
1.2.2.Maintenance Criteria	4
1.2.3.Performance Criteria	4
1.2.4.Trade-offs	5
1.3.References	5
2. Software Architecture	6
2.1.Overview	6
2.2.Subsystem Decomposition	6
2.3.Architectural Styles	8
2.3.1.Layers	8
2.3.2.Model View Controller	8
2.4.Hardware / Software Mapping	8
2.5.Persistent Data Management	9
2.6.Access Control and Security	9
2.7.Boundary Conditions	9
2.7.1.Execution	9
2.7.2.First Run	10
2.7.3.Termination	10
3. Subsystem Services	11
3.1.Design Pattern - Façade	11
3.2.View Management Subsystem	11
3.3.Game Management Subsystem	16
3.4.Game Objects Management Subsystem	20
4. Low Level Design	28
4.1.Object Design Trade-offs	28
4.1.1.Performance vs. Complexity	28
4.1.2.Responsiveness vs. Memory	29
4.2.Final Object Design	29
4.3.Packages	30
4.4.Class Interfaces	30
4.4.1.Keyboard Listener	30
4.4.2.Mouse Listener	31

1. Introduction

1.1.Purpose of the System

OtoParker is a top viewed 2D computer game. The main purpose of the game is very much like any other car parking game. Simply, it is to park your car in a designated area without colliding with other objects. In addition to that, our game integrates modern day games' features like power-ups and destructible environments. So, it's technically one step further.

1.2.Design Goals

The main goal for a computer game is first, to entertain the player. To achieve this crucial task, we need to focus on the little details which are not necessarily directly noticeable at a first glance.

This section details the design goals of the system such as end user criteria, maintenance criteria, performance criteria and the trade-offs that come with our chosen way of implementation. The following points are extensions of the non-functional requirements of our analysis report.

1.2.1.End User Criteria

Target User Base: When we observe other successful games available today, we see that they have one common attribute. It is the range of the user base. As an example, Candy Crush^[1] is very successful because a kid and an old person can both enjoy it equally.

Ease of learning: When starting the game for the first time, the user will not know how the controls or the game mechanics work. So, it is important to provide the user with a smooth learning curve. First couple levels will be designed in such a way that it teach the controls and the mechanics of the

game. After these simple levels, the player will be pushed towards greater challenges to increase the fun factor.

Ease of use: Most computer games use common controls these days to increase the ease of use. Menu navigations are done by the mouse pointer and the character (in our case the car) is moved by the arrow keys. Moreover, our planned Upgrade Car screen (available on the Analysis Report) is similar to other upgrade screens of other games. Since most users are already familiar with these designs the game will be easy to learn and play.

1.2.2.Maintenance Criteria

Extensibility: In the lifecycle of a software program, it is important to maintain the ability to add or remove features. As object oriented software engineering principles imply, the first goal in development is to create a highly maintainable software.

Portability: In today's software world, everything is changing rapidly. Some years ago, IBM's processor architecture was very popular, then Intel came and people realized the need for a portable programming language. Now, ARM is slowly replacing Intel's x86 architecture so our software should be able to run on the ARM architecture in the future. So, we decided to use the Java Programming Language to keep our code portable for future hardware.

1.2.3.Performance Criteria

Smooth Graphics: In order for a game to be engaging, it must provide smooth frame rates. Especially in our game, players will need all the details they could get to navigate the car into the parking spot with ease.

Input Response Time: Inputs from the keyboard while playing the game needs to be very precise because the players will be very frustrated if the car crashes due to input lag.

1.2.4.Trade-offs

Portability - Performance & Memory: In this project, we will be using the Java Programming Language, which is known for its ability to produce binaries which can run on any processor architecture. However, this results in all the applications being run in an interpreter called the Java Virtual Machine. When compared to native languages like C++, programs written with Java use more resources while doing less. On the flip side, coding our game with Java will decrease the development time due to the broad range of libraries available and the game will be running on any Java compatible desktop device.

Simplification - User Base: In our end user criteria, we detailed the target user base to be very broad. When a game targets a broad range, it requires over simplification of some game mechanics. For example, we could have integrated features like tire grip while turning a corner. If the player was turning too fast, the car could have drifted a little bit. Or we could have used very detailed and intensive graphics which would have required a powerful desktop computer, which many users don't have. So, we decided to keep the game simple and have a broader user base.

1.3.References

[1] https://en.wikipedia.org/wiki/Candy_Crush_Saga

2. Software Architecture

2.1.Overview

In this section, we will start detailing the composition of our system from a higher level. We have decided to divide our system into smaller subsystems. This will result in more maintainable and simpler parts coupled to each other rather than a huge chunk of code. In our project, we will be using the Model View Controller design architecture since we think it suits best for a computer game like ours.

2.2.Subsystem Decomposition

In this section, the smaller subsystems of our software will be shown in detail. Later in section 3, all these small subsystems will be detailed further. While making these subsystem divisions, we have considered the extensibility and modifiability of our code. If we want to change a part of the game, we should be able to do it without impacting the other parts. In the following page, the diagram for these subsystems are shown. We have three subsystems: View Management, GameManagement and GameObjectManagement.

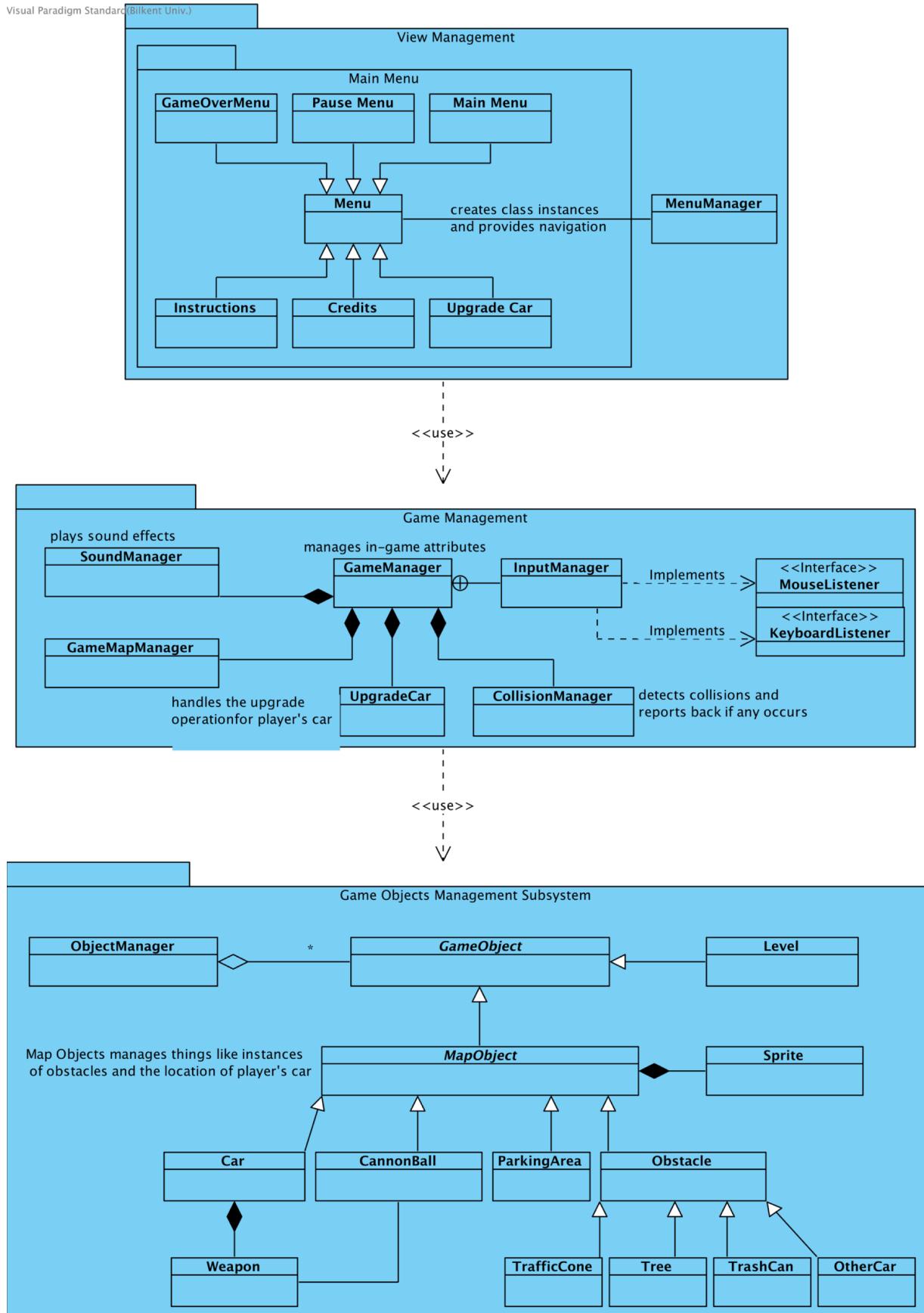


FIGURE 2.2.1 - SUBSYSTEMS

2.3.Architectural Styles

2.3.1.Layers

In our decomposition, we have decomposed our system into three layers.

We have the user interface layer on top which manages the layers below.

View Management layer can interact with these layers to provide required functionality.

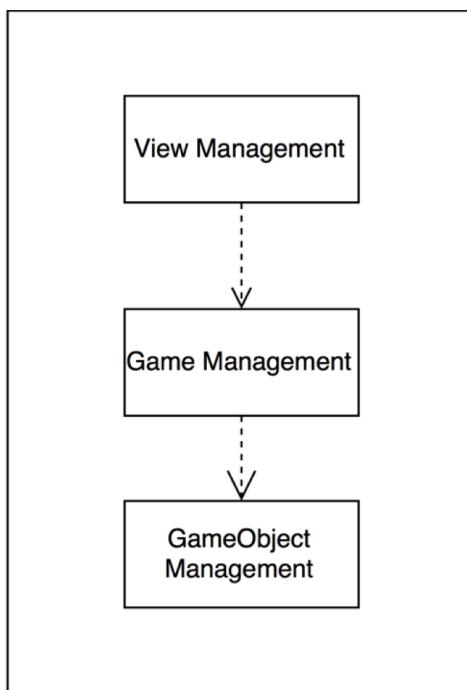


FIGURE 2.3.1.1 - THREE LAYERS OF THE SYSTEM

2.3.2.Model View Controller

In our game, we decided to use the modal view controller design

architecture. This design architecture will help us with the isolation of these layers from the player.

2.4.Hardware / Software Mapping

Our game OtoParker will be implemented using the Java Programming Language.

We will be using Java 8 in this implementation since it has beneficial features for

the development of the game. As for the hardware configuration, OtoParker requires a basic mouse and keyboard.

With these hardware and software requirements, any user with a desktop computer running Windows/macOS/Linux and Java installed should be able run our game.

2.5.Persistent Data Management

Our game will store the game state such as the applied car upgrades, the levels passed and starts earned in a comma separated file. This is a universal file format where there are many available Java libraries such as the Apache commons. Using this file format will help us in the development cycle. These files will be mapped to the home directory of the user on a Unix based operating system and the Documents folder on Windows.

2.6.Access Control and Security

Our game OtoParker will not require any access controls or security measures since the saved game data does not include any private information about the player.

2.7.Boundary Conditions

2.7.1.Execution

OtoParker will not require any software other than Java Runtime Environment installation on the players' computers.

2.7.2.First Run

On the first run, OtoParker will search for a saved game state, however when it doesn't find any, it will only create the required folders to save on exit.

2.7.3.Termination

There are a few options to close the game OtoParker. There will be an Exit Game button (as shown in the Analysis Report) on the Pause Menu and Game Over Menu. Also, the player can use the close window button of the operating system. (X located in top right in Windows and the red dot located in top left in macOS)

3. Subsystem Services

3.1.Design Pattern - Façade

This design pattern aims to provide a unified and high level interface which many objects in the program could use. As an example from our project, View Management Subsystem in Section 3.2 uses the Menu class as the start point. Other Menu classes inherit from this class to provide functionality. This enables developers to reflect changes in the project in a more dynamic way.

Another example is the MapObject class from Section 3.4. This class will provide functionality like collision boxes to different objects in the game. Therefore, the façade design pattern further improves the maintainability and reusability of our code.

3.2.View Management Subsystem

View Management subsystem has eight main classes that provides an interface between the user and the system. MenuManager controls all the menu objects and determines which one is going to be displayed.

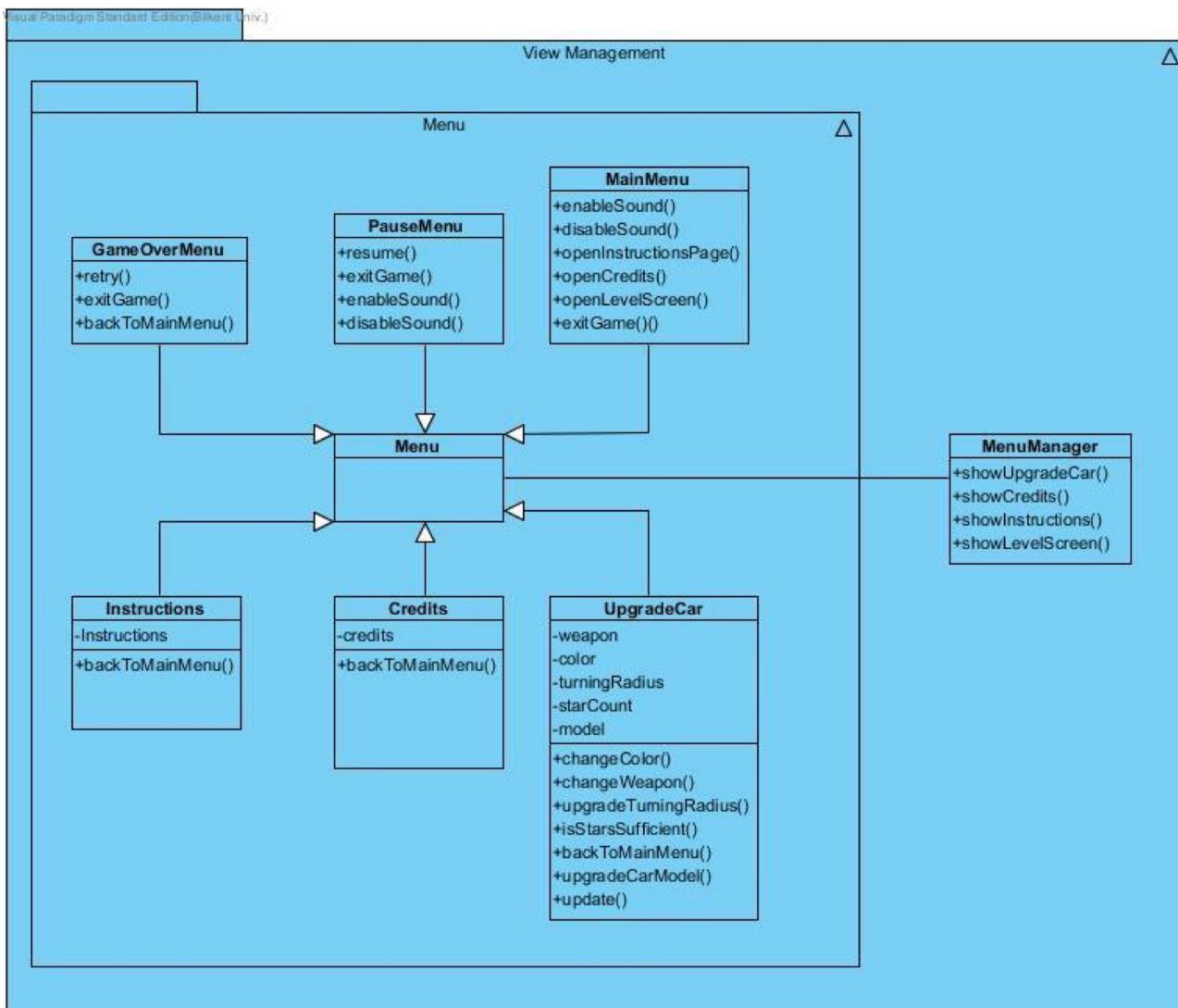


FIGURE 3.2.1 - VIEW MANAGEMENT SUBSYSTEM

MenuManager:

Handles requests of menus

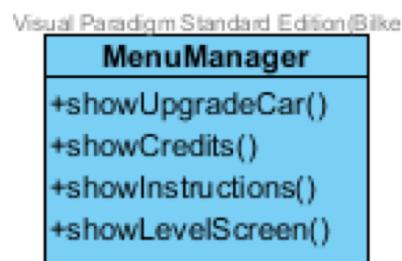


FIGURE 3.2.2 MENU MANAGER CLASS

Operations:

public void showUpgradeCar(): Handles opening upgrade car menu.

public void showCredits(): Handles showing credits.

public void showInstructions(): Opens Instructions screen.

Menu: This class is parent class of all menu classes.

MainMenu:

This class is first running class. When players open the game this class creates Main Menu and user make choice. Also players can reach main menu during the game with using other menus.

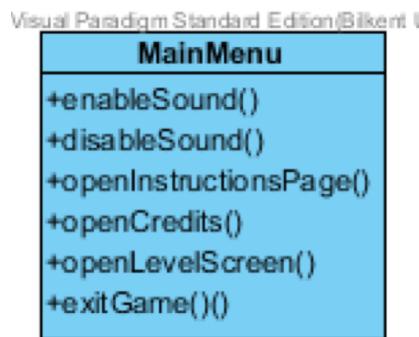


FIGURE 3.2.3 - MAIN MENU CLASS

Operations:

public void enableSound(): This methods invokes game manager for enabling sound.

public void disableSound(): This methods invokes game manager for disabling sound.

public void openInstructionsPage(): Invokes menu manager for opening instructions.

public void openCredits(): Invokes menu manager for opening credits.

public void openLevelScreen(): This option invokes game manager for opening level screen. After this level screen players can start the game.

public void exitGame(): Player exits game with pressing this option.

PauseMenu: User can exit game or enable/disable sound with this menu.

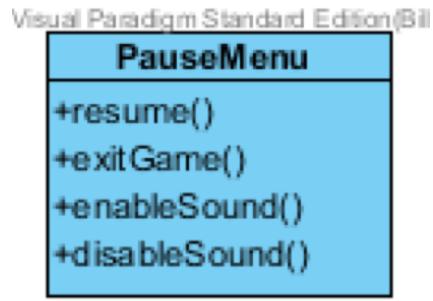


FIGURE 3.2.4 - PAUSE MENU CLASS

Operations:

public void resume(): User can resume game with this option.

public void exitGame(): This method closes game.

public void enableSound(): This methods invokes game manager for enabling sound.

public void disableSound(): This methods invokes game manager for disabling sound.

GameOverMenu:

After user crash or finish map this menu appears.



FIGURE 3.2.5 - GAME OVER MENU CLASS

Operations

public void retry(): This method invokes game manager for starting same map again.

public void exitGame(): This method closes the game.

public void backToMainMenu(): This method invokes menu manager for opening main menu.

Instructions: Shows instructions screen. There are instructions for game in this screen.

Credits: Shows credits screen. Players can see contributors in this screen.

UpgradeCar: This class shows upgrade car menu to players. Players can upgrade or change their cars.

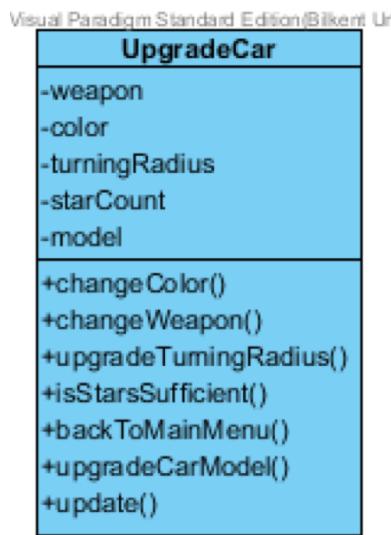


FIGURE 3.2.6 - UPGRADE CAR CLASS

Attributes

private Weapon[] weapon: List of weapons.

private String[] color: List of colors.

private int[] turningRadius: List of turning radiuses.

private int[] starCount: Storages star count of user.

private Car[] model: List of car models.

Operations

public void changeColor(String color): This method changes color of car.

public void changeWeapon(Weapon weapon): Changes weapon or adds weapon to car.

public void upgradeTurningRadius(int turningRadius): Changes turning radius.

private boolean isStarsSufficient(): This method controls that is stars count of user sufficient for desired upgrade.

public void backToMainMenu(): This method invokes menu manager to open main menu.

public void upgradeCarModel(Car model): Changes car model.

public void update(): Updates current car with new upgrades.

3.3.Game Management Subsystem

This subsystem implements the game logic with the main controllers. The vital features of the game such as collision detection, upgrades, game map design are all handled here.

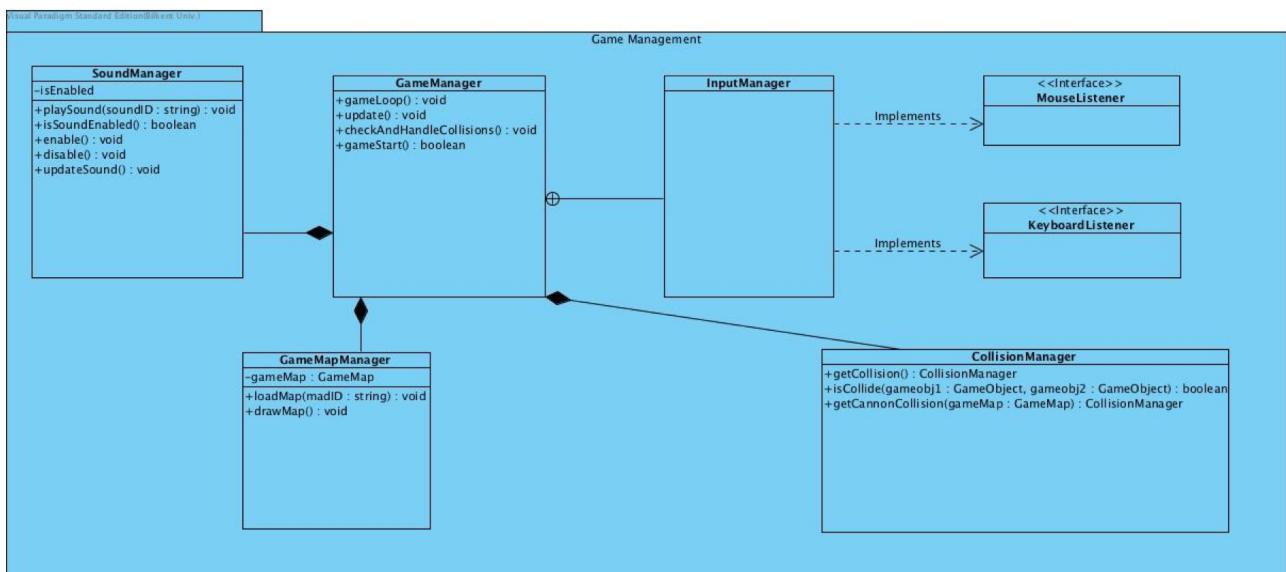


FIGURE 3.3.1 - GAME MANAGEMENT SUBSYSTEM

GameManager:

The class that starts the game and also runs the game loop of the game. During the game it checks for the collisions and updates the current game according to the changes that occurred during the game.



FIGURE 3.3.2 - GAME MANAGER CLASS

Operations

public void gameLoop(): Runs the base loop that initiated with the game that creates the gameplay

public void update(): Updates the changes that occurred in the game that generated by the user.

public void checkAndHandleCollisions(): During the game checks if there is any collisions and if there are, uses CollisionManager to handle them.

CollisionManager:

Decides and handles the collisions that occurred during the game play. It also checks whether the player has made a successful shot with the weapon on the car.

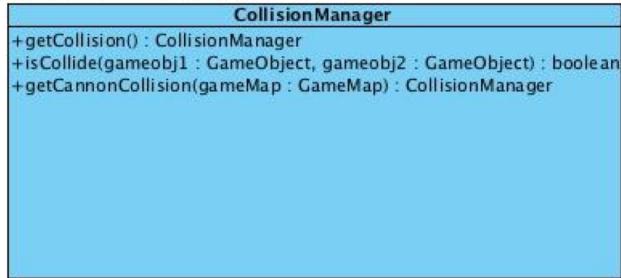


FIGURE 3.3.3 - COLLISION MANAGER CLASS

Operations

public CollisionManager getCollision(): This method gets the collision that occurred in the game and allows game to decide with it.

public boolean isCollide(GameObject gameObj1, GameObject gameObj2): Checks if there is a collision between two game objects

public CollisionManager getCannonCollision(GameMap gameMap): When a player makes a shot, this method gets the place of collision from the map.

GameMapManager:

Controls dynamic changes such as shot obstacles in current level's gameplay map.

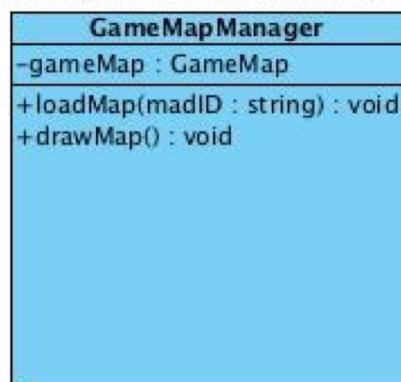


FIGURE 3.3.4 - GAME MAP MANAGER CLASS

Attributes

GameMap gameMap: The object that the game play screen contains

Operations

public void loadMap(String mapID): Loads the selected game map into the screen

public void drawMap(): The method that draws the map.

InputManager: Handles the inputs that comes from the mouse and keyboard.

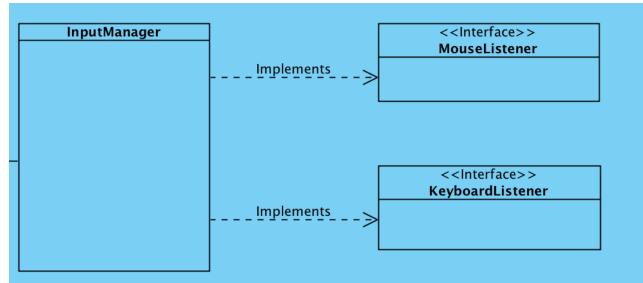


FIGURE 3.3.5 - INPUT MANAGER CLASS

Sound Manager:

This class will instantiate all the in-game objects for game to start.

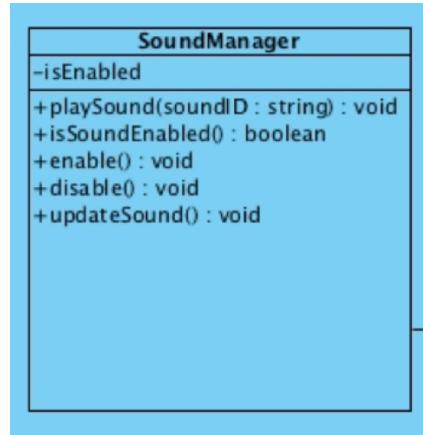


FIGURE 3.3.6 - SOUND MANAGER CLASS

Attributes

private bool isEnabled: true if the sound effects are enabled

Constructors

public SoundManager(): It will read the configuration file if the sound was enabled on the last exit or not to set the isEnabled property

Methods

public void playSound(String soundId): plays sound with ID
public boolean isSoundEnabled(): return true if sound enabled
public boolean enable(): set isEnabled to true
public void disable(): set isEnabled to false
public boolean updateSound(): retrieves and stores list of sound IDs.

3.4.Game Objects Management Subsystem

Main objects that user interacts with are placed in this subsystem, such as map objects which user sees during playing, or levels and stars which are parts of the game. All game objects are inherited from an abstract class named Game Objects and all map objects are grouped under MapObjects class.

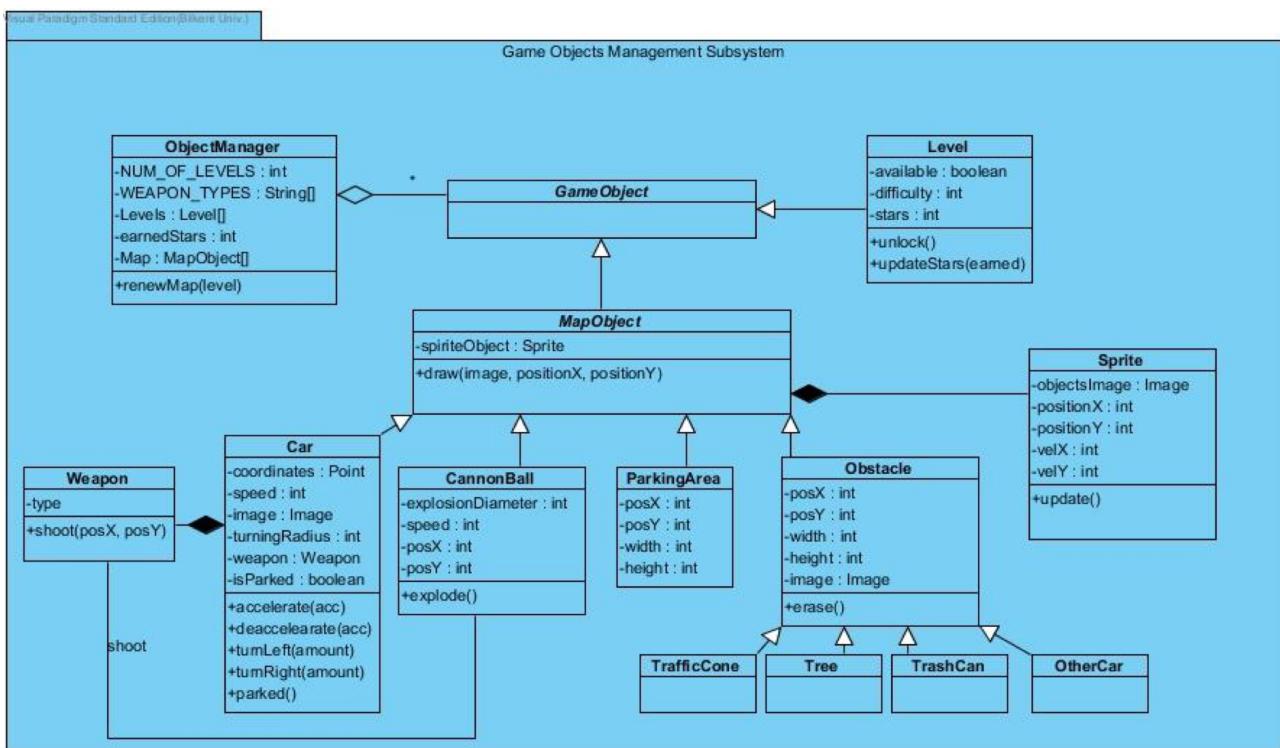


FIGURE 3.4.1 - GAME OBJECTS MANAGEMENT SUBSYSTEM

Object Manager:

This class will instantiate all the in-game objects for game to start.

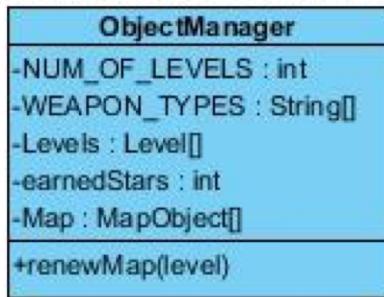


FIGURE 3.4.2 - OBJECT MANAGER CLASS

Attributes

private int NUM_OF_LEVELS: This attribute shows which level is the level object.

Primarily it will be equal to 10, since we design to implement 10 levels.

private String[] WEAPON_TYPES: This will list the weapon names in the game.

private Level[] Levels: This attribute will contain the level instantiations.

private int earnedStars: Number of stars gained by the player will be stored in this attribute.

private MapObject[] MapObjects: Map Object instantiations will be stored in this attribute.

Constructors

public ObjectManager(): This default constructor will instantiate the attributes to their defaults.

public ObjectManager(Level[] l, int stars): This constructor will instantiate the class with a given levels array and earned stars amount, in case the game saved before.

Methods

public void renewMap(Level l): Using this method, the map objects of the class will be renewed, for example when starting a new game, or after unlocking a new level.

Level:

Visual Paradigm Standard Edition(Bilkent)

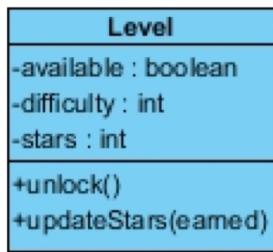


FIGURE 3.4.3 - LEVEL CLASS

Attributes

private boolean available: This will show whether the level is unlocked by the player or not.

private int difficulty: This is the difficulty, or degree, of the current level object, which will be between 1 and NUM_OF_LEVELS.

private int stars: This will show the number of stars gained in that level.

Constructors

public Level (int difficulty): With a given difficulty, a level object will be created.

Methods

public void unlock(): This method will change unlocked attribute to true if user gains access to the level.

public void updateStars(int earned): This method will update the number of stars of the level object with a given number of stars.

MapObject:

Visual Paradigm Standard Edition(Bilkent Univ.)

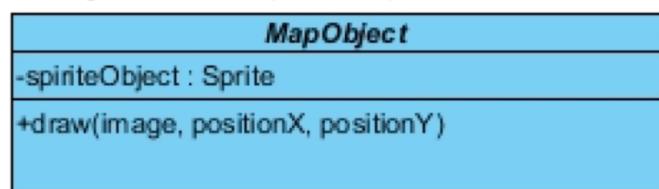


FIGURE 3.4.4 - MAPOBJECT CLASS

Attributes

private Sprite spriteObject: This is an instance of the Sprite class for every map object in the game.

Methods

public void draw(Image I, int posX, int posY): Using this method, every Map Object will be drawn on the map using the positions of X and Y coordinates.

Sprite:

This class is will be used for updating the position of an object given the values of velocity and positions at X and Y coordinates.

Visual Paradigm Standard Edition/Bilkent Univ

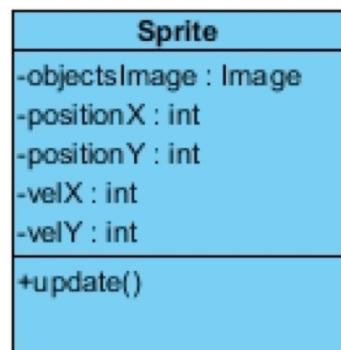


FIGURE 3.4.5 - SPRITE CLASS

Attributes

private Image objectsImage: This attribute is the in-game appearance of the given object in Image type.

private int positionX: X coordinate values of the object.

private int positionY: Y coordinate values of the object.

private int velX: Velocity of the object in X direction.

private int velY: Velocity of the object in Y direction.

Constructors

public Sprite (Image img, int posX, int posY, int velX, int velY): Given the attributes, a sprite object will be created.

Methods

public void update(): Location of the object will be updated according to given velocities, once at a time.

Weapon:

This class defines the weapon of the player's car.

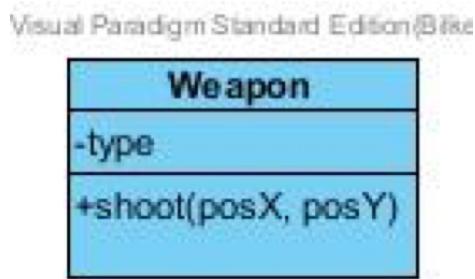


FIGURE 3.4.6 - WEAPON CLASS

Attributes

private String type: This will specify the type, also the name of the given weapon.

Constructors

public Weapon(String type): With the specified name, a weapon is created.

Methods

public void shoot(int posX, int posY): With the specified direction, a cannonball is shot. Its parameters will be same as the CannonBall's directions.

CannonBall:

This is the class for objects shot from the weapon.

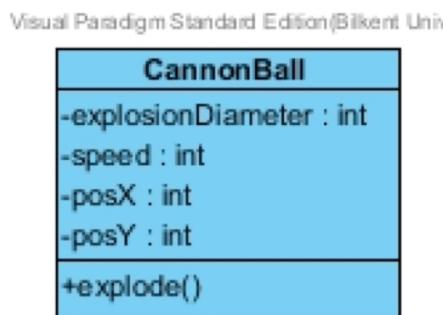


FIGURE 3.4.7 - CANNONBALL CLASS

Attributes

private int explosionDiameter: This attribute specifies the diameter of the collision when it touches some other object.

private int speed: Speed of the cannonball.

private int posX: Value of the cannonball's direction on X coordinate.

private int posY: Value of the cannonball's direction on Y coordinate.

Constructors

public Cannonball(int diameter, int posX, int posY): Given the directions, a cannonball object is created with a certain diameter.

Methods

public Point explode(): This method will delete the cannonball from the map, will show explode animation and also to delete the object in the specified coordinates, will return its final point of coordinates.

ParkingArea:

This class is for specifying the coordinates of parking slot that player must park the car in.

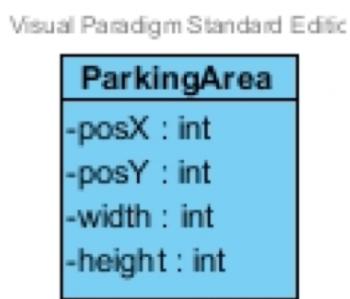


FIGURE 3.4.8 - PARKINGAREA CLASS

Attributes

private int posX: Specifies the position of the area on X coordinate.

private int posY: Specifies the position of the area on Y coordinate.

private int width: Width of the parking area.

private int height: Height of the parking area.

Constructors

public ParkingArea(int pX, int pY, int h, int w): With given dimensions, a parking area object is created.

Car:

This class depicts the main object that are directed by the player, car.

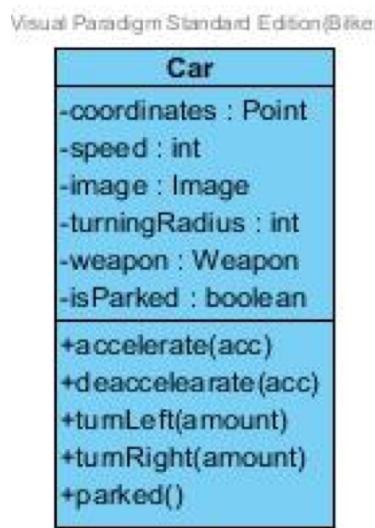


FIGURE 3.4.9 - CAR CLASS

Attributes

private Point coordinates: Coordinates of the car on the map.

private int speed: Speed of the car.

private Image: Image of the car.

private int turningRadius: This attribute shows that how much car moves when player wants to go right or left directions.

private Weapon weapon: Weapon of the car.

private boolean isParked: This attribute will show the state that if the car is parked or not.

Constructors

public Car(Point c, Image i, Weapon w): The car will be created on the map's given coordinates with given image and wanted weapons.

Methods

public void accelerate(int acceleration): This attribute will increase the car's speed by given acceleration.

public void deaccelerate(int deacceleration): This attribute will decrease the car's speed by given de-acceleration.

public void turnLeft(int amount): This attribute will change the car's direction to the left by the given amount.

public void turnRight(int amount): This attribute will change the car's direction to the right by the given amount.

public boolean parked(): Returns the car's isParked attribute.

Obstacle:

This class defines the obstacle objects on the map. This class have 4 children which will all have same attributes: TrafficCone, Tree, TrashCan, OtherCar.

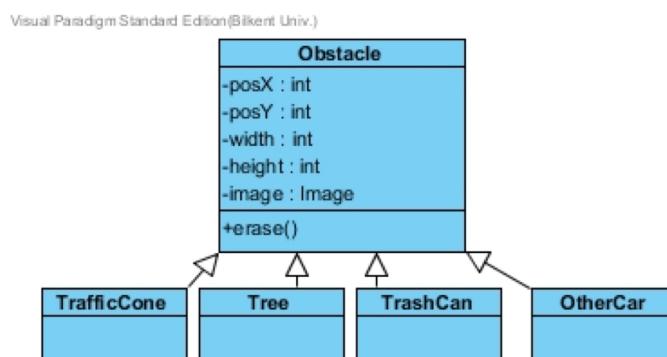


FIGURE 3.4.10 - OBSTACLE CLASS

Attributes

private int posX: Specifies the position of the obstacle on X coordinate.

private int posY: Specifies the position of the obstacle on Y coordinate.

private int width: Width of the obstacle.

private int height: Height of the obstacle.

private Image: Image of the obstacle.

Constructors

public Obstacle(int pX, int pY, int h, int w, Image i): With given specifications, the obstacle object is created.

Methods

public void erase(): This method is for erasing the obstacle from the map, for example when a weapon fires at the obstacle.

4. Low Level Design

4.1.Object Design Trade-offs

4.1.1.Performance vs. Complexity

Our main purpose during game play is to provide high performance, to achieve that components in the game should be dynamically changing. Therefore, our object design focused on the implementation of a Game Manager, to be able to handle the changes and reflect them instantly. For example, during game play, the player may want to change the model of his car, or buy a new one by pause menu. When he continues to game, his changes should be reflected on the current game. Therefore, to be able to handle these kind of changes a layered hierarchy is required, so that View Management layer may reflect the changes that are handled by the other layers independent of the different changes to game objects. Therefore view management and game management layers are separated at the cost of complexity, because a more complex system will result in a more complicated implementation process.

4.1.2.Responsiveness vs. Memory

User interface of the game designed to work robustly, meaning that all the screens in the game should be easily changeable. For example, from pause screen the player should be able to change to update or other scenes instantly. Therefore, rather than drawing over and over a single panel when needed, we designed several View Management classes to be instantiated at the beginning of the game and they will be seen when user navigates, instantly. Therefore these view layer objects will occupy more space in the memory but will work more robustly.

4.2.Final Object Design

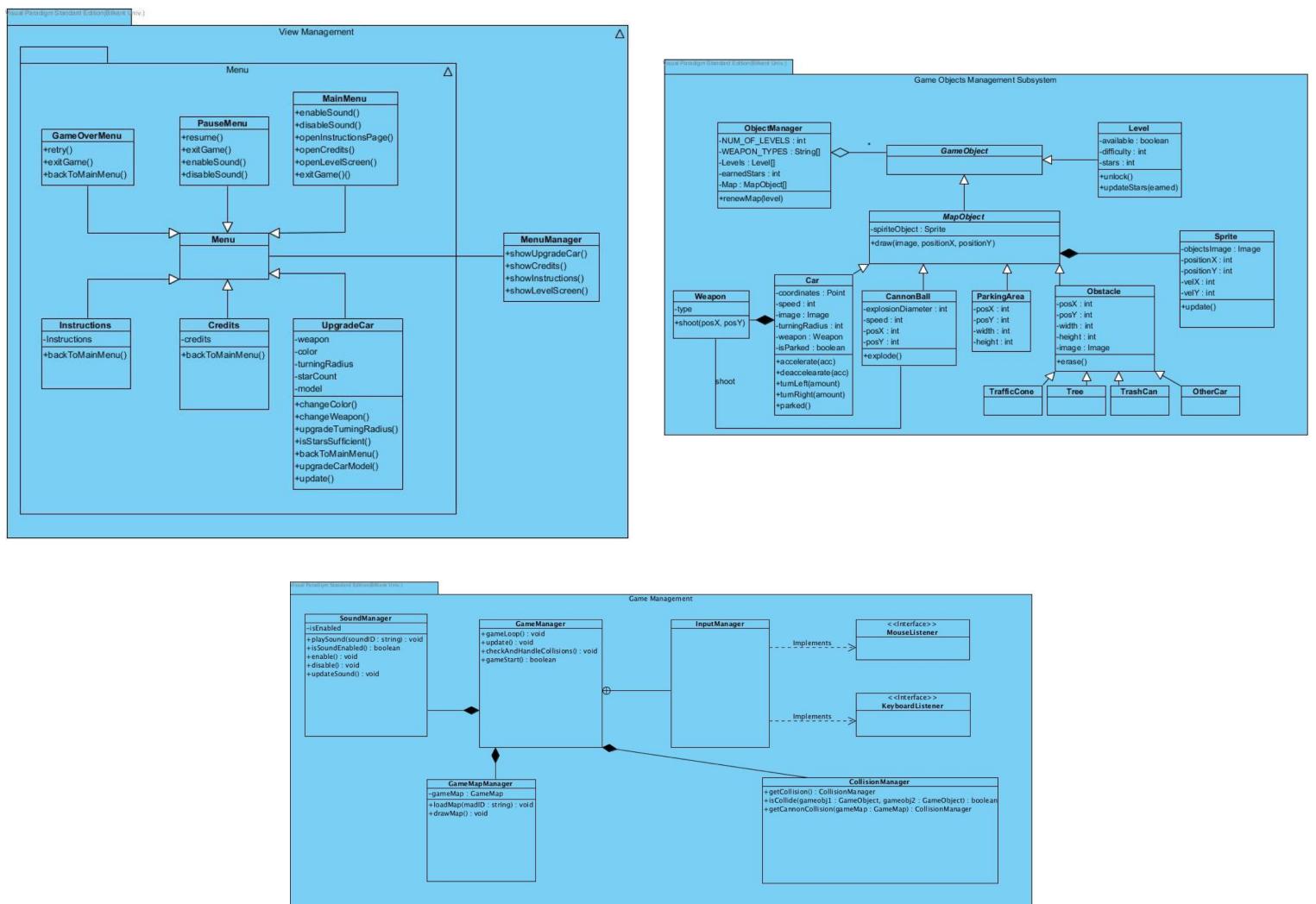


FIGURE 4.2.1 - FINAL OBJECT DESIGNS

4.3.Packages

We have three packages for three subsystems. We used packages for dividing our code to smaller subsystems. These subsystems provide us to controlling and designing systems with less complexity. Small and more related parts are more preferable than entire code.

Our first package is View Management Subsystem. It contains menus and menu manager. Also There is another package in this package which menu package. Menu package contains menu screens. These screens are credits, instructions, main menu, pause menu, game over menu, upgrade car menu. Another package includes Game Management Subsystem. It contains manager classes such as game manager, sound manager, input manager, game map manager, collision manager. Last package is Game Objects Subsystem. In this package there are game objects class hierarchy. There is a game object and their inherited classes like map object, car, obstacles, cannonball. Obstacles class also has some inherited classes like traffic cone, tree, trashcan and other car.

4.4.Class Interfaces

Interfaces are collections that consists of abstract methods. A class simply uses the methods that it needs to fulfill its function and keeps other methods empty bodied. In OtoParker we have two interfaces that are being used in "InputManager" class.

4.4.1.Keyboard Listener

KeyboardListener interface allows InputManager class to get, identify and process keyboard input in the game. User's movements in the game and user's weapon usage is provided through this interfaces methods. In the menu keyboard events do not create any change. However "esc" button provides a "back button" function so if "esc" is pressed this keyboard event

creates an event that takes user to the previous page, if it has used in the pause menu user returns to the game play.

4.4.2.Mouse Listener

MouseListener interface allows InputManager class to get and use the mouse actions in the game. Mouse actions are mainly used in menu screen and upgrade screen. Also level selection screen and pause menu screen use the mouse actions. Mainly used mouse action is click in the game. Mouse actions do not create any changes in the gameplay so MouseListener methods are not involved in the gameplay process, however in the pause menu mouse actions are used.