CS 319 - Object-Oriented Software Engineering

System Design Report

Umut Akös

Oğuz Liv

Mithat Orhan

Bilal Siraj

Contents

1- Introduction

1.1 Purpose of the system

Overtake is a classic arcade car game with a twist. We have taken an all-time favorite arcade game of a car speeding down a highway and avoiding obstacles along the way, trying to survive for as long as possible, and added new elements, adding to the challenge and reviving this classic. More obstacles, power-ups, and a range of new vehicles make Overtake a more engaging version of it's monochrome ancestor. Overtake uses a simple yet complete user interface making it easily learnable for the user, and utilizes dynamic game difficulty balancing to ensure the difficulty is always at a sweet spot.

1.2 Design Goals

Adaptability: Overtake will be implemented using the Java programming language allowing for the game to be played on multiple operating systems, without us, the developers, having to make any changes to the source code.

Extensiblity: The current design of Overtake is such that it does not allow much room for changes. The reason for this is because we had a clear idea of what we want the game to be and that is a simple reinterpretation of a classic arcade game. This means we did not anticipate any need to add other features than what is already implemented in the game. That would be to deviate from our initial objective.

Reliability: The game is planned to be completely bug free, having all boundary cases well tested and leaving no room for error. However, as with all software, it is still possible for anomalies to pop up.

Usability: As the description of the purpose of the system stresses, the game is simple, meaning it allows for the user to get acquainted with and comfortable with the controls, features and objectives in the game very easily. To make such a simple game appealing, we have also made sure the user interface and gameplay experience are easy on the eye.

1.3 Trade Offs

Modifiability and Reusability: The way Overtake is designed does not make changes very easy. Since all game logic is implemented within one class, the controller, any updates to the model classes, or addition of new models, will be a challenge to integrate with the existing system. Reusability of the source code is not a goal we had for our project.

Functionality: Overtake is not the most complex of games and does have a rather limited functionality. The game only implements the basic concepts of an arcade car game with slight changes to the objectives and challenges.

2. Software Architecture

2.1 Overview

In order to reduce coupling among subsystems, system will be divided to sustainable subsystems. For architectural style, we have main controller so that decomposed system will be synchronized properly.

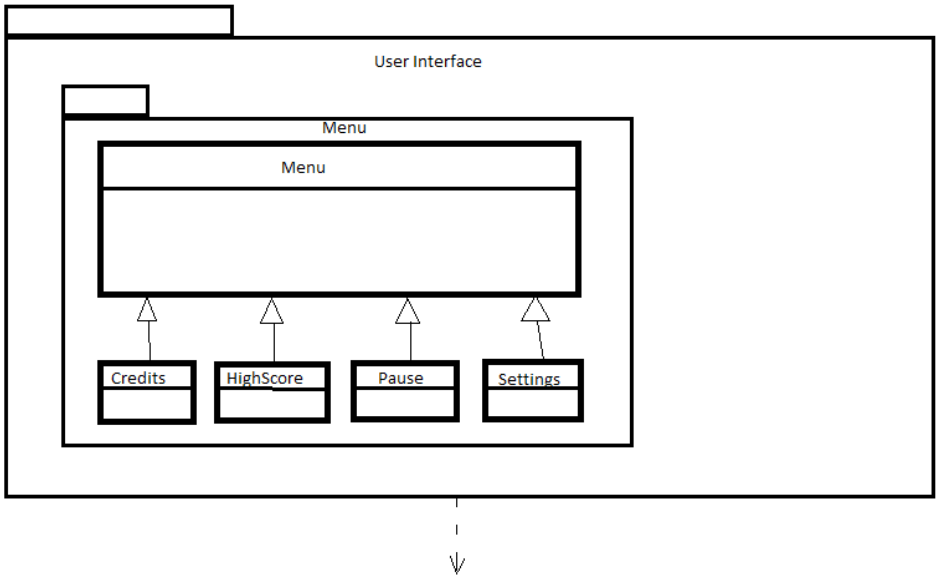
2.2 Subsystem Decomposition

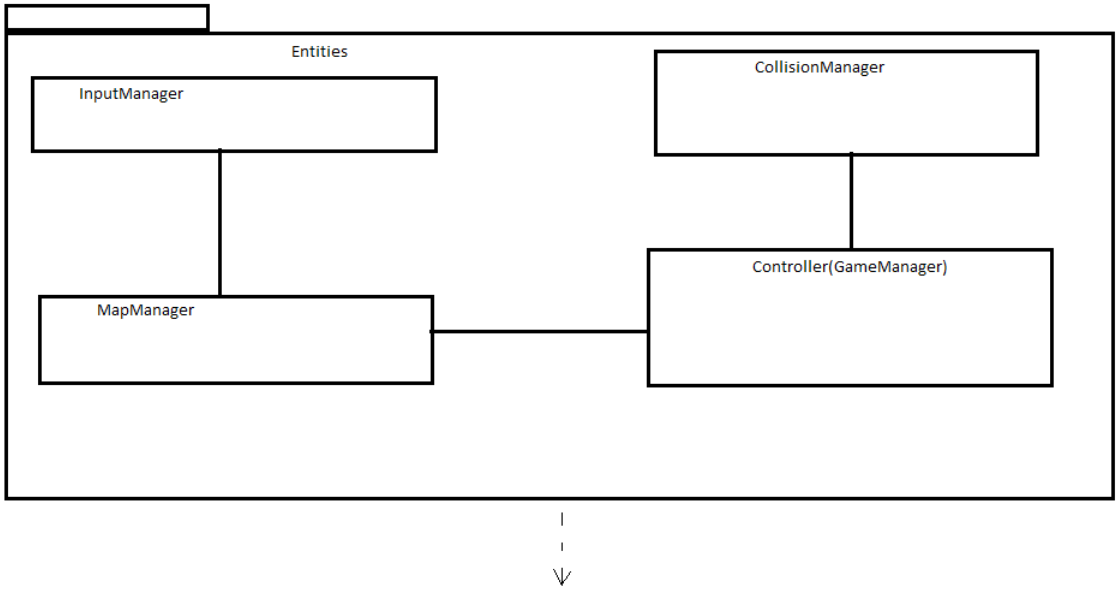
Our system is seperated into synchronized and independent parts for proper architecture and organization. Thus, we will be able to control our system much healthier. Moreover there will be considerable performance increase and ease on modification and extension via this architecture. In this way, both functional and non functional requirements division are playing significant role for high quality software.

Our subdivided system is shown below figure. Main general divisions, subsystems are User interface, Management and Entities(synchronized with controller). As it is shown, there is connection between each subsystem. To identify connection between Management and Entity subsystems, controller is emphasized in each system. This provides enhanced synchronization among them. There is many inheritance usage in our systems, which will lead that if any error is occured, only its parent will be responsible from this specific error.

In detail, it is shown that children classes who are performing similar tasks, are gathered for optimization of systems. This organization will ease the subsystems in terms of controlling and modification if needed.

Consequently, subsystems of our project is is investigated to implement our goals effectively and with high quality. High cohesion and loosely coupling are implemented to reach our goals as well. This caused flexible and simple system that can be modified or upgraded easily when needed.


User Interface
Menu
Menu
Credits
HighScore
Pause
Settings


Entities
InputManager
CollisionManager
Controller(GameManager)
MapManager

2.3. Archirectural Styles

2.3.1 Layers

As it is mentioned above, our system is divided into three subsystems which are User interface, Management and Entities. We will use these three subsystems as layers to improve the quality of hiearchy. For instance, User interface layer will be at the top of hierarchy so that other subsystems will not be able to use User interface. On the other hand, Entities layer will be at the bottom of the hiearchy because entity object are brought and used together in our game. The architecture of layers is shown figure below.

2.3.2 Controller

In our project design, we will have one main controller as it is implemented in 2.2. This main controller will be able to modify classes which are settings, collectiable(and its children), vehicle(and its children). For example, main user interface will not able to be modified by controller. This hierarchical main controller will be also responsible by starting, pausing game,checking and updating highScores, generating and updating obstacles, enemies and players. By our hierarchy design, if we modify the user interface, this wont effect controller and other subsystems.

2.4 Hardware / Software Mapping

We will use Java language during our game implementation. Hence JDK 1.7 or above will be used during software process. For inputs we will have KeyListener and MouseListener so keyboard and Mouse will be required. System requirements will be minimal as we are implementing the game through simple in Java with no complicated game engine. We will be using photoshop cs6 for our designs. Also java platform will be used for designs.

Internet will not be required for our game and we will store highScores in txt file. Thus for storage, our java implementation will read from txt file.

2.5 Persistant Data Management

We will store the Project data in the hard disk drive, for highScores our desing will not be through database because the simplicity of our data which are scores and players. Image and model datas will be stored independently so that they will be easily modified.

2.6 Security

Since the game is not required internet, it is resictricted in specific pc. Secondly, for data corruption in specific pc, by our hierarchical design, will be much easier to not effecting noncorrupted datas.

2.7 Boundary Conditions

- Initializing the game

No install will be required. The game will be initialized by single jar file.

- Termination

Quit game button in the interface will be provide user to terminate the game. If user wants to quit during game, controller will warn the user if he/she is sure or not. Also standart "x" termination button will exist.

- Error

In case of errors which are caused by user's pc, current player's at that time game data will be lost.

3. Subsystem Services

In this section we will provide the detailed information about the interfaces of our subsystems.

3.1. Design Patterns

Façade Design Pattern: Façade design pattern is a structural design pattern which proposes that developers can easily manage a subsystem from a façade class since the communication between outside of this subsystem is performed only by this class. This pattern, provides maintainability, reusability and extendibility since any change in the components of this subsystem can be reflected by making changes in the façade class.

In our design we used façade pattern in two subsystems: Game Management and Game Entities. In Game Management subsystem our façade class is Controller which communicates with the components of the Game Management subsystem according to the requests of User Interface subsystem. For our Game Entities subsystem our façade class is again Controller class which handles the operations on entity objects of our system, according to the needs of Game Management subsystem.

3.2 – User Interface Subsystem Interface

User Interface subsystem provides graphical system components for our project. In addition, it also manages transition between panels which are constructed for different selections in main frame screens. The reference of User Interface Subsystem to other subsystems is provided by mainFrame class which is considered as an interface.

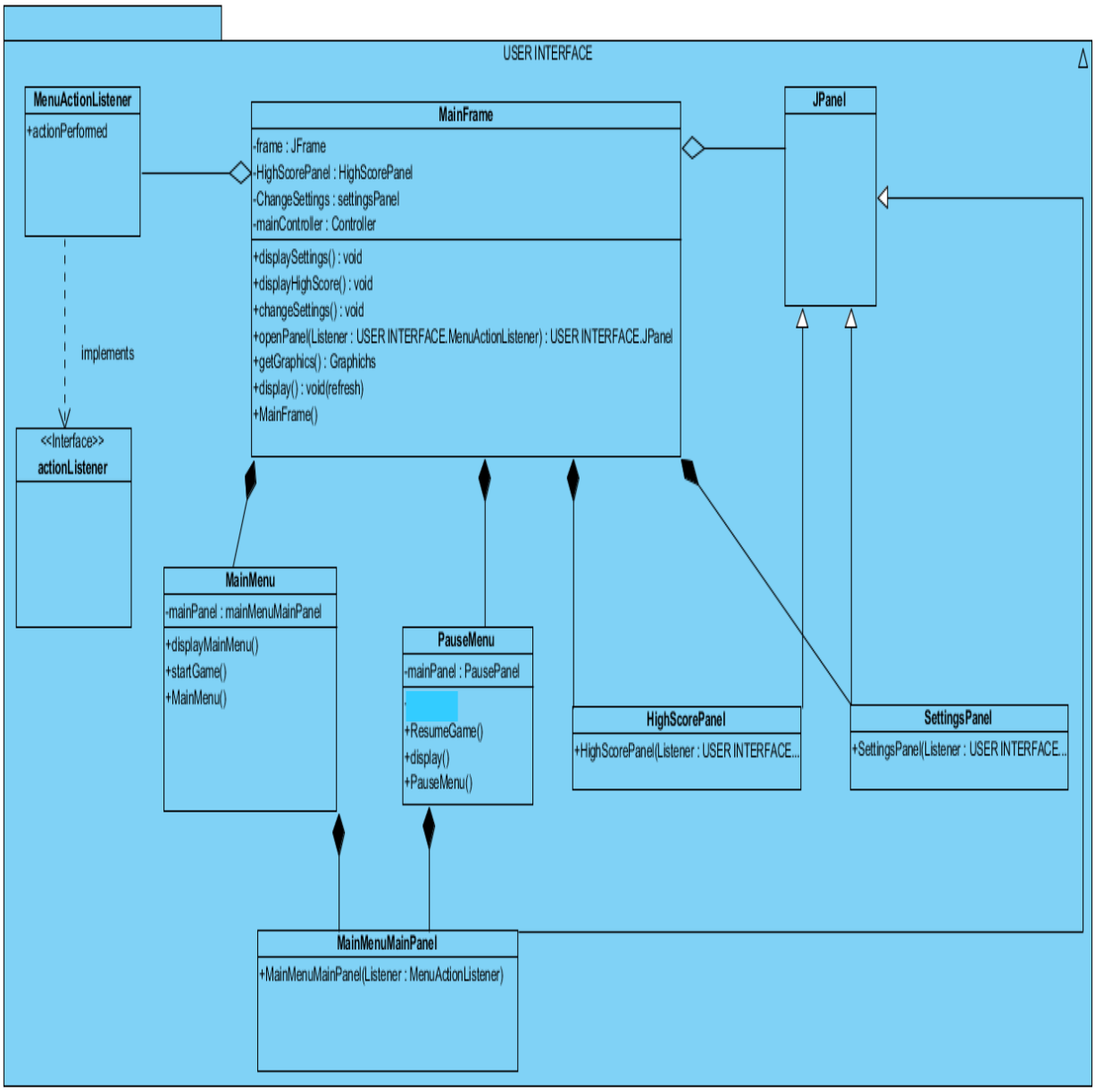
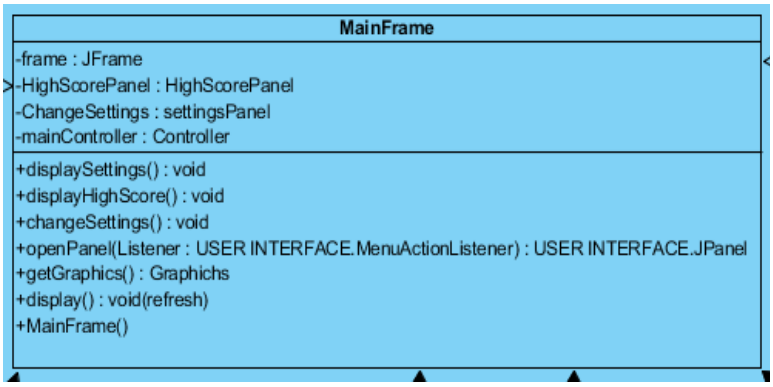


Figure-1(User Interface Subsystem)

MainFrame Class



Attributes

private Jframe frame: This is the frame which provides display all visual context for us.

private HighScorePanel: This panel shows the highest score of the user.

private ChangeSettings settingsPanel: This Panel or Jpanel is used in graphical user interface to show to user setting or change setting menu on the screen.

- This property is constructed by ChangeSettings class with its components like buttons, labels etc…

private mainController controller: This mainController type property of MainFrame class provides reference to GameManagement subsystem, when Play Game selection is received from graphical user interface.

- This property is constructed by mainController class with its components like buttons, labels etc…

Constructor

mainFrame(): constructs a frame that our panels can fit into it properly.

Methods

public void displaySettings(): It represents change settings panel for the main frame.

public void displayHighScore(): It provides the high score display and user can see the high score in this screen.

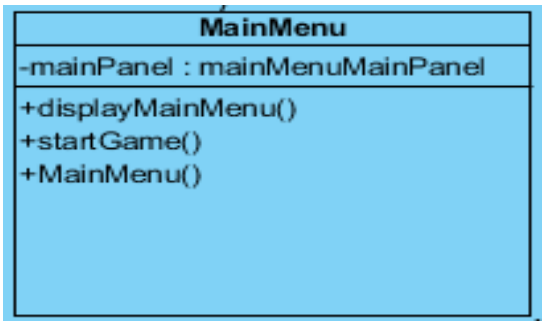
public void changeSettings(): It holds together all components which provides changing settings for the user.

public Jpanel openPanel(): It provides the opening new panel for us. It provides panel to the main frame as a basic operation.

public Graphics getGraphichs(): returns a graphics context for drawing to an off-screen image.

public void display(): It initializes the game and entity objects which are coming from the controller are displayed by this method.

Main Menu Class



Attributes

private mainMenuPanel mainPanel: this is the main panel that main menu to be displayed in terms of graphical user interface

Constructors

public MainMenu(): constructs a main menu by modifying mainMenuPanel.

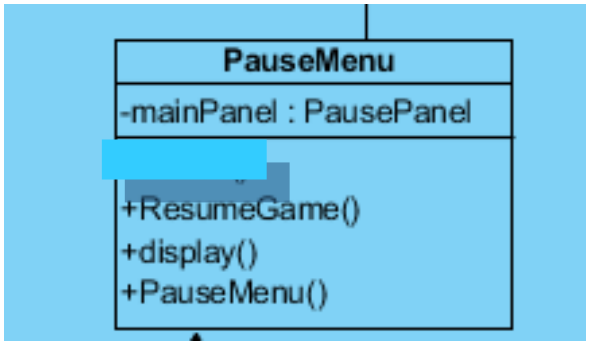
Methods

public void displayMainMenu(): this method puts the mainMenuPanel into the MainFrame in order represent it on the screen.

public void startGame(): starts the game logic and loads the game components.

- **Following methods are inherited from parent MainFrame class**
  - displayHighScore()
  - displaySettings()
  - openPanel (MenuActionListener e, Jpanel p)

Pause Menu Class



Attributes

private PausePanel mainPanel: this is the pause panel that pause menu to be displayed in terms of graphical user interface

Constructor

public PauseMenu(): constructs a main menu by modifying PausePanel.

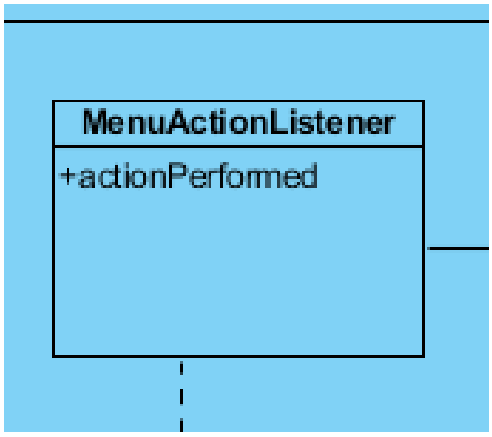
Methods

public void display (): this method puts the PausePanel into the MainFrame in order represent it on the screen.

public void ResumeGame(): in the state of stopping all objects, this methods starts stopped objects moving according to game logic.

- **Following methods are inherited from parent MainFrame class**
  - displayHighScore()
  - displaySettings()
  - openPanel (MenuActionListener e, Jpanel p)

MenuActionListener class



public void actionPerformed(ActionEvent e): overrides actionPerformed() method from ActionListener interface.

- HighScorePanel, ChangeSettingsPanel, MainPanel classes are not detailly expained, they instantiate requested panels and ActionListener places them on the main frame.

3.3 Game Management Subsystem Interface

Game management Subsystem Interface, our controller objects are combined to manage the actual game dynamics and game logic. We have 6 classses and we have just one controller class and other classses are property classses. In Figure-1, controller class can be seen and setting and GameInfo are subclasses of the Controller and Runnable,KeyListener and MouseListener are interface.

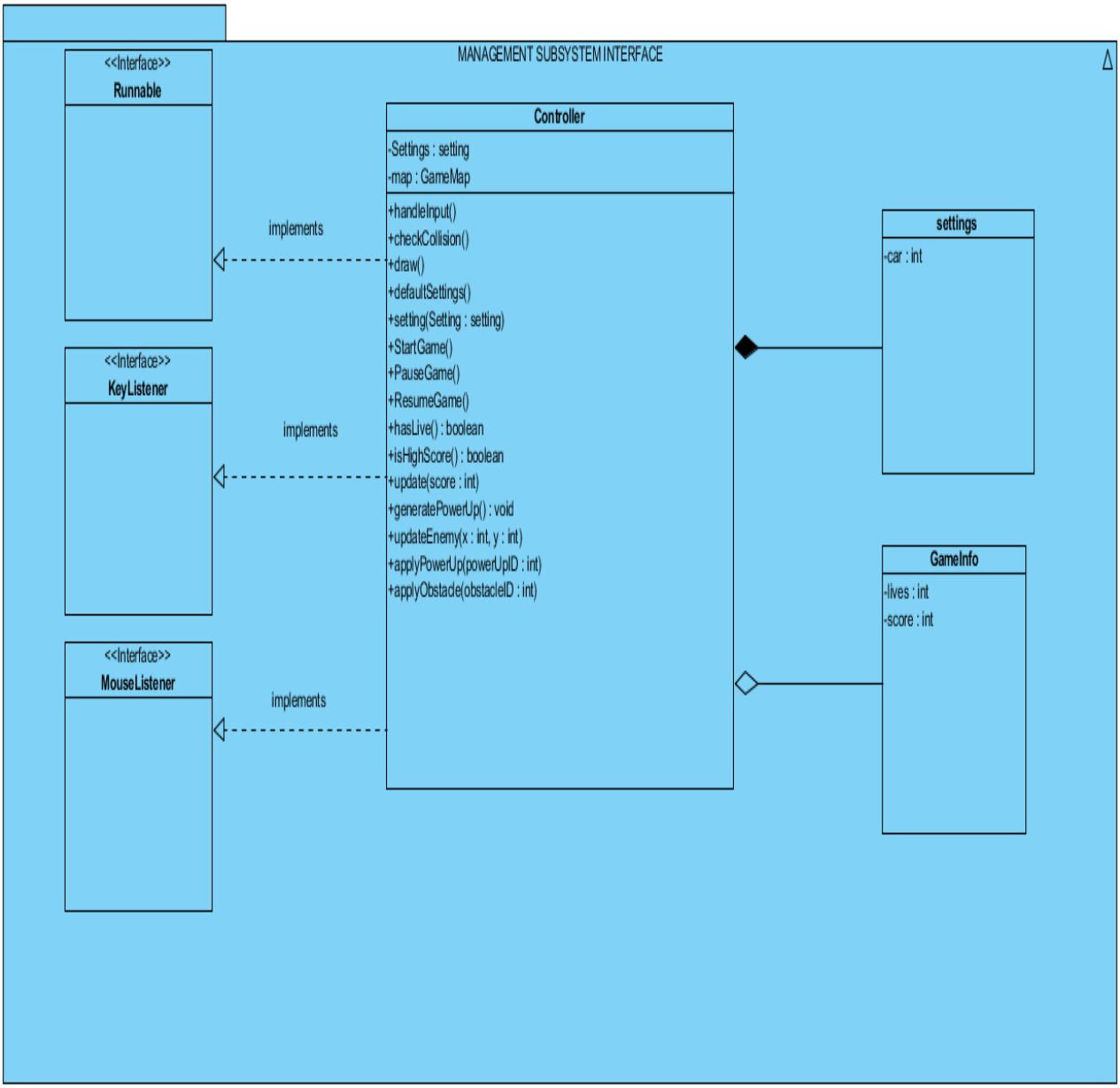
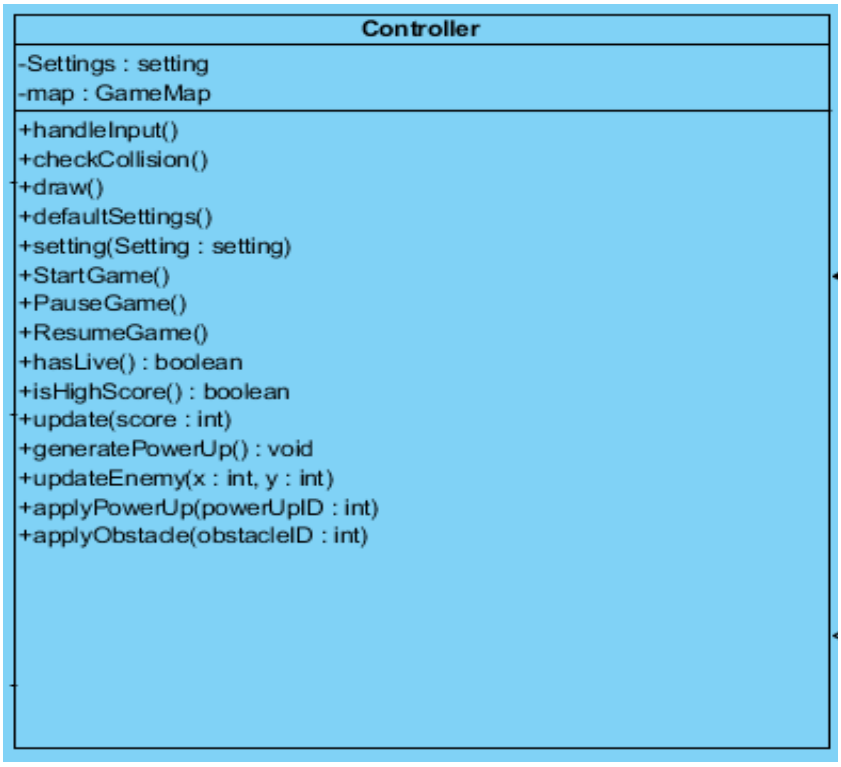


Figure-2(GameManagement Subsystem Interface)

Controller Class



This class is the Façade class of the GameManagement subsystem. User Interface subsystem send some requests and according the these requests, it performs operations properly and this class implements runnable interface which provides running the game loop as a thread.

Attributes:

private Setting settings: this attribute provides holding settings of the game, like car type so user can play his/her car which he select in the beginning of the game.

private GameMap map: this attribute is a map object, controller class associates with proper methods of GameMap class.

Methods:

public boolean handleInput(): it checks the user playing, in the playing, it checks that user can whether the obstacle or not.

public boolean checkCollision(): it checks the wheter the car crash with other enemy car or obstacle. If it does, it returns true, otherwise false and his/her score is increases.

public void draw(): it provides the drawing the enemies and other obstacles during the game and add them the main frame.

public void defaultSettings(): it resets the all settings which is set by user and it restore all settings.

public Setting setting(): it enables possibility of choosing the settings in the beginning of the game.

public void StartGame(): it starts the new game and all of the old game are not stored so all game which are played are deleted.

public void PauseGame(): it poused the game so Timer in the code stops so user see the panel which has label is naming as Continue if he/she want to continue, should click it and Timer starts again.

public void ResumeGame(): after the pousing, it provides continutiy for the user so user can resumer the game remaining of his game.

public boolean hasLive(): it checks the user live during all the game and if it returns true, user can continue the game but if it returns false,user will see another screen.

public boolean isHighScore(): In the game all scores are saved but it is for just one play so while user is playing the game, this method checks his/her score and compare with his/her other scores.

public void update(): update the game while user are playing it but it just for his/her score because for power-ups and enemies updating, game has another methods.

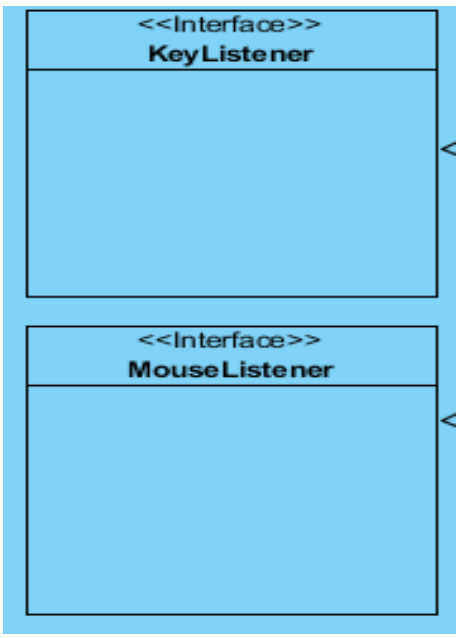
public void generatePowerUp(): it generates the power-ups. In the game user can find different power-ups so we should generate them, this method provides it.

public void updateEnemy(): it generates the enemies. We have not one type enemies. Therefore in the game user can see different enemies so like power-ups we should generate them, this method provides it.

public void applyPowerUp(): After generating power-ups, it applies the given power-ups to the system.

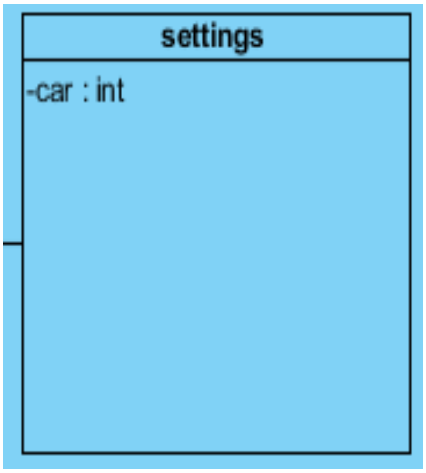
public void applyObstacle(): After generating obstacles, it applies the given obstacle to the system.

Listener Classes



These class are designed to detect user actions while he/she are playing the game. Our game is played by using keyboard so we need the take user action by using KeyListener class so if user want to exit or pause the game, he/she should use the mouse so we need also MouseListener class and moreover user needs to use mouse in the beginning of the game.

Setting Class



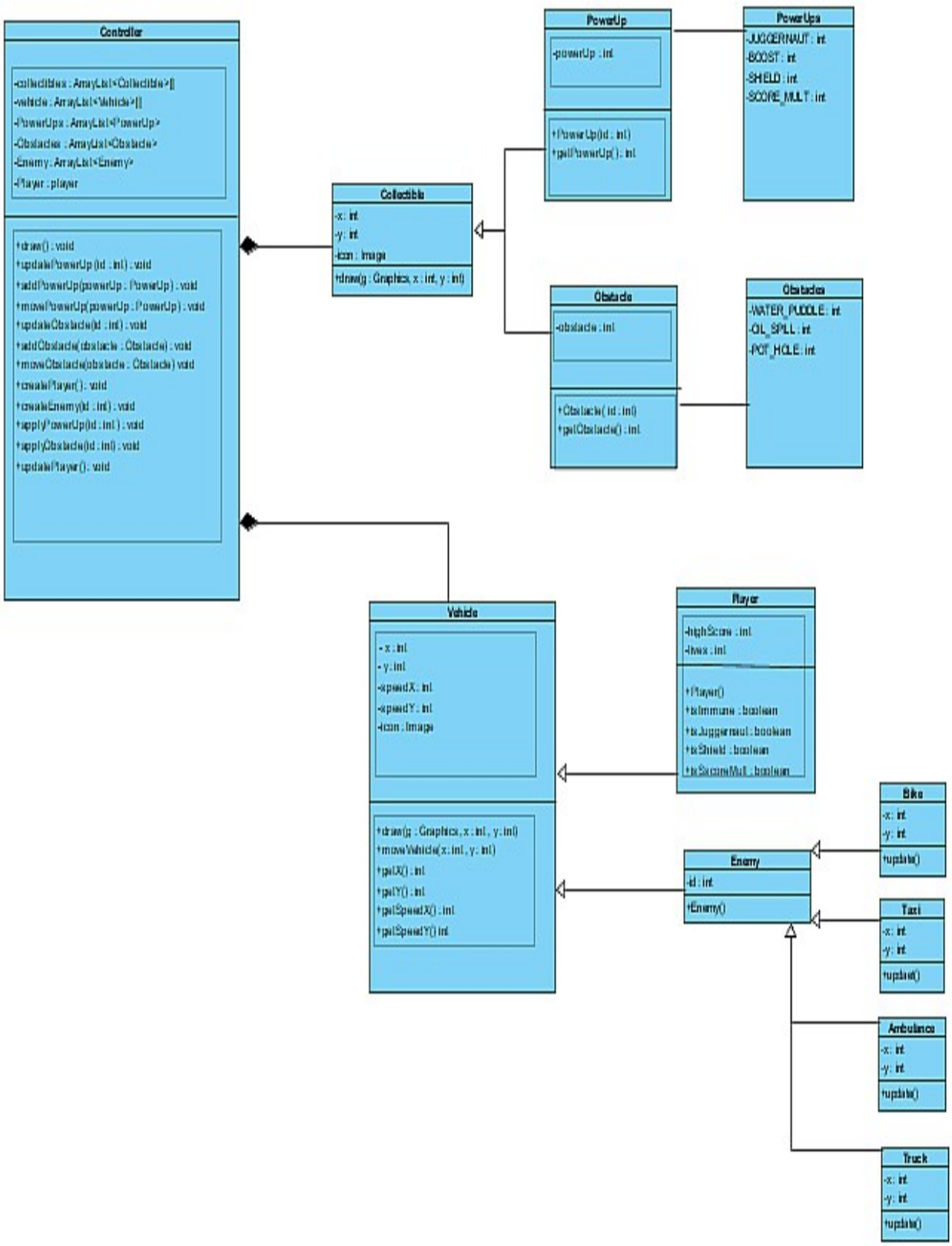
Setting class provides settings car number in the game,actually we can named it as a difficulty of the game because in the game, user can set the number of cars in the game and if car number is more, game would be more difficult.

GameInfo Class

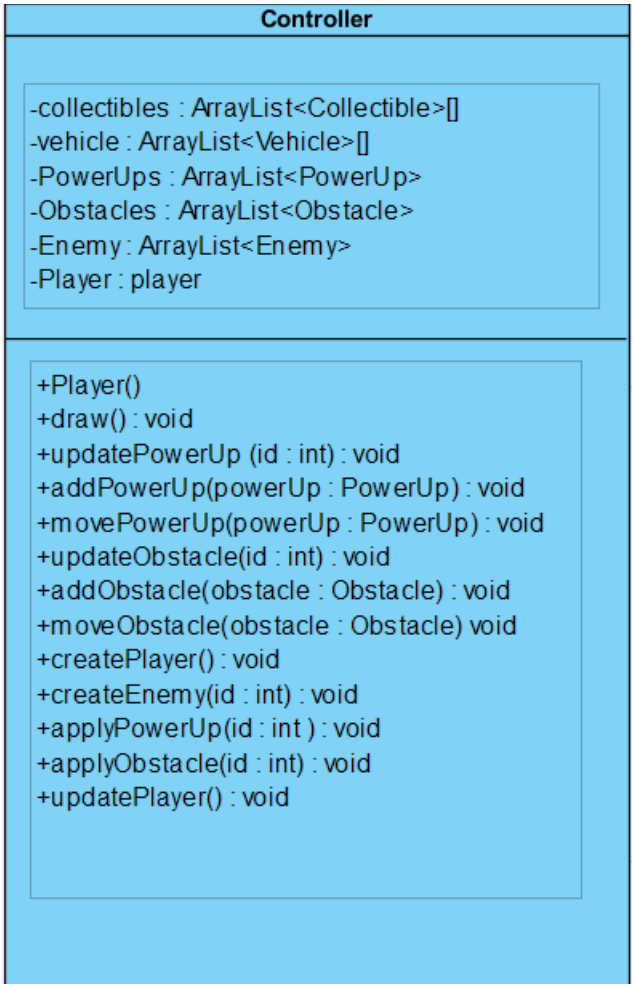
GameInfo class gives some information about players gaming. One of them is his/her remainning live and other one is score. So thanks to the this class, user can learn remainning lives and score and he/she does not need to exit the game if he/she click the gameInfo, he/she will learn.

3.4 Game Entities Subsystem Entities

Entities subsystem is the susbsystem that contains domain specific objects. This subsystem consists of 10 classes. Player car and enemy car inherit from Vehicle abstract class. Also additional features of game inherit from Collectible abstract class. Façade class of this subsystem is Controller class, which controls the objects of the system and modify them according to game. Because of our Façade design pattern, dividing systems into subsystems minimizes the association of these subsystems. In the figure Game Entities subsystem visualized. Each class of will be specifically explained in this section.


Controller
-collectibles : ArrayList<Collectible>[]
-vehicle : ArrayList<Vehicle>[]
-PowerUps : ArrayList<PowerUp>
-Obstacles : ArrayList<Obstacle>
-Enemy : ArrayList<Enemy>
-Player : player
+draw() : void
+updateIPowerUp (id : int) : void
+addIPowerUp(powerUp : IPowerUp) : void
+moveIPowerUp(powerUp : IPowerUp) : void
+updateObstacle(id : int) : void
+addObstacle(obstacle : Obstacle) : void
+moveObstacle(obstacle : Obstacle) void
+createPlayer() : void
+createEnemy(id : int) : void
+applyPowerUp(id : int) : void
+applyObstacle(id : int) : void
+updatePlayer() : void
Collectible
-x: int
-y: int
-icon : Image
+draw(g : Graphics, x : int, y : int)
PowerUp
-powerUp : int
+PowerUp(id : int)
+getPowerUp() : int
PowerUps
-JUGGERNAUT : int
-BOOST : int
-SHIELD : int
-SCORE_MULT : int
Obstacle
-obstacle : int
+Obstacle(id : int)
+getObstacle() : int
Obstacles
-WATER_PUDDLE: int
-OIL_SPILL : int
-POT_HOLE : int
Vehicle
- x : int
- y : int
-speedX: int
-speedY: int
-icon : Image
+draw(g : Graphics, x : int, y : int)
+moveVehicle(x: int, y: int)
+getX() : int
+getY() : int
+getSpeedX() : int
+getSpeedY() int
Player
-highScore : int
-lives : int
+Player()
+isImmune : boolean
+isJuggernaut : boolean
+isShield : boolean
+isScoreMult : boolean
Enemy
-id : int
+Enemy()
Bike
-x: int
-y: int
+update()
Taxi
-x: int
-y: int
+update()
Ambulance
-x: int
-y: int
+update()
Truck
-x: int
-y: int
+update()

Controller Class



- Controller class is the Façade class of this subsystem, thus, this class's operations modifies and creates entities of the game. Also communication of the objects occurs here.

Attributes

private ArrayList<Collectible> collectibles: this array list references to the all collectibles of the game.

private ArrayList<Vehicle> vehicle: this array list holds all vehicle objects of the game.

private ArrayList<PowerUp> PowerUps: this array represents all power ups on the screen.

private ArrayList<Obstacle> Obstacles: this array represents all obstacles on the screen.

pivate ArrayList<Enemy> Enemy: this array holds all enemy objects on the screen.

private Player player: Since there will be only one player, this attribute represents player itself.

Methods

public void draw(): draws all the vehicle and collectible objects on the main panel and updates them.

public void updatePowerUp(int id): this methods updates power up's coordinates which are on the screen according to their ID's.

public void addPowerUp(PowerUp powerUp): this method adds created power up to PowerUp array list in order to be on the screen soon.

public void createPowerUp(PowerUp powerUp): creates new power up

public void updateObstacle(int id): this method updates coordinates of the obstacles which are on the screen

public void addObstacle(Obstacele obstacle):this method adds created obstacle to the obstacle list in order to be presented on the screen.

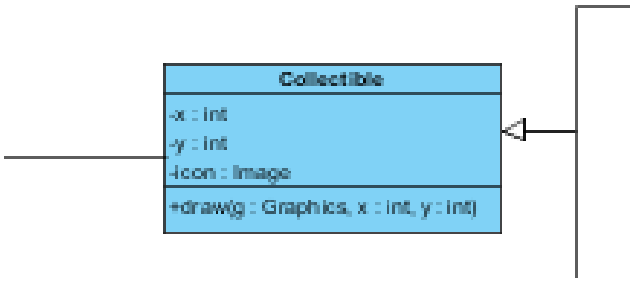
Public void createObstacle(Obstacle obstacle): creates new obstacle.

public void createEnemy(int id): creates new enemy.

public void createPlayer(): constructs new player.

Public void updatePlayer(): updates players controls as user input taken in terms of moving.

Collectible Class



- This class is an abstract class for image and 2D coordinates for collectibles.

Attributes

private int x: this attribute is x-axis value for collectibles.

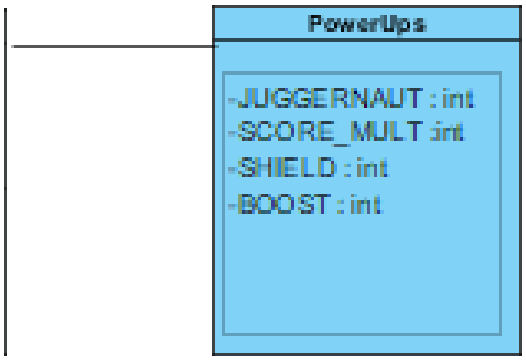
private int y: this attribute is y-axis value for collectibles.

private Image icon: this attribute is for how a collectible represented on the screen.

Methods

public void draw(Graphics g, int x, int y): This method draws the collectible object to proper location on the main screen. For instance, generating water puddle on the screen but any random place.

PowerUps



Attributes

private int JUGGERNAUT

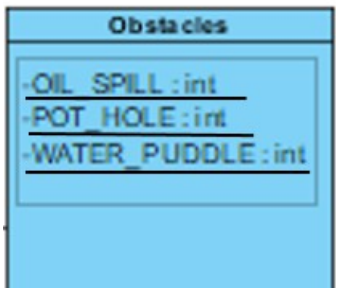
private int SCORE_MULT

private int SHIELD

private int BOOST

- These attributes define the kinds of power ups.

Obstacles



Attributes

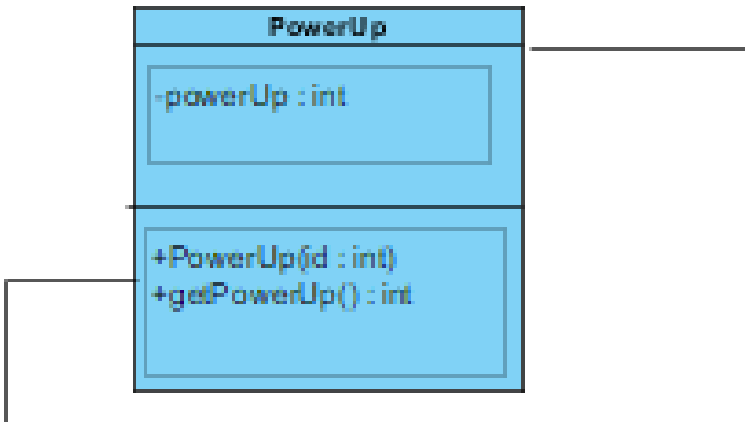
private int OIL_SPILL

private int POT_HOLE

private int WATER_PUDDLE

- These attributes define the kinds of obstacles.

PowerUp Class



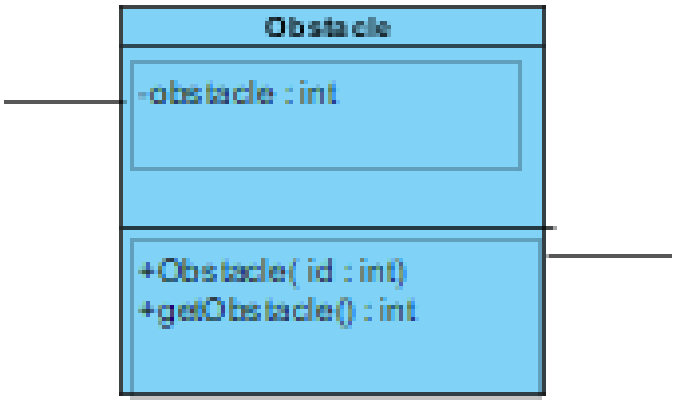
Attributes

private int powerUp: this attribute holds the id as the kind of power up.

Constructors

public PowerUp(int id): constructs a power up object according to its kind.

Obstacle Class



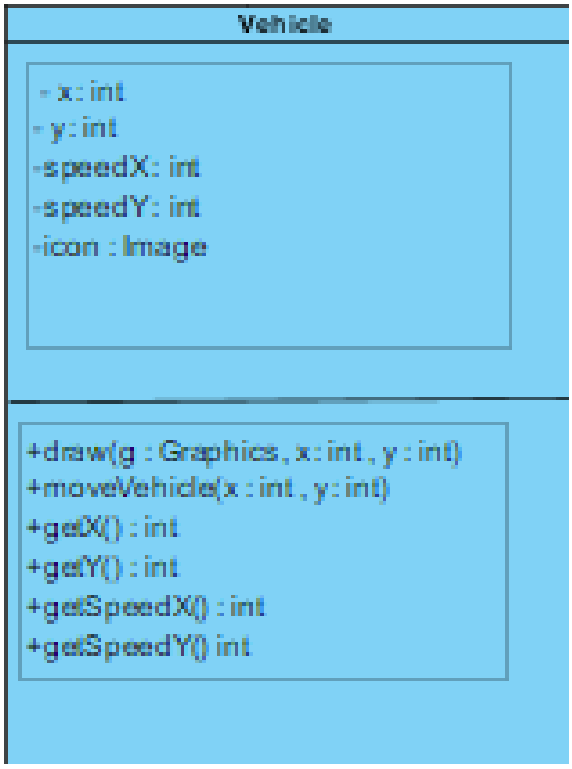
Attributes

private int obstacle: this attribute holds for the kind of obstacle.

Constructors

Public Obstacle(int id): constructs an obstacle object accordingly its kind.

Vehicle Class



- This abstract class changes the objects attributes, in other words, as objects states changed, this class represents those states features in terms of game.

Attributes

private int x: this attribute holds the x-axis value for the vehicles.

private int y: this attribute holds the y-axis value for the vehicles.

private int speedX: how fast the vehicle's x-axis value changes.

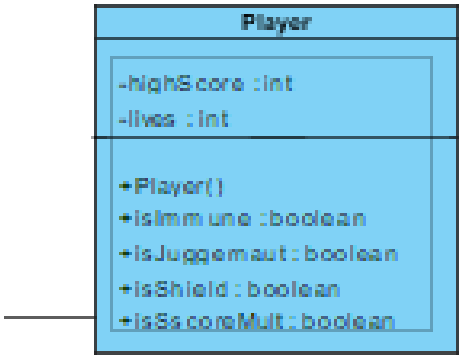
private int speedY: how fat the vehicle's y-axis value changes.

Methods

public void draw(Graphics g, int x, int y): This method draws the vehicle object to proper location on the main screen. For instance, generating enemy cars on the screen but any random place. Also locates players location according to user.

public void moveVehicle(int x, int y): updates coordinates of the vehicles according to game.

Player Class



Attributes

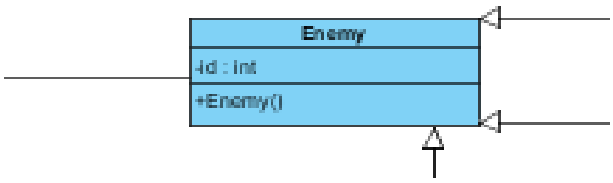
private int highScore: score of the player that he makes during the game.

private int lives: remainin lives of the player during the game.

Constructor

public Player(): constructs player object and initializes its attributes.

Enemy Class



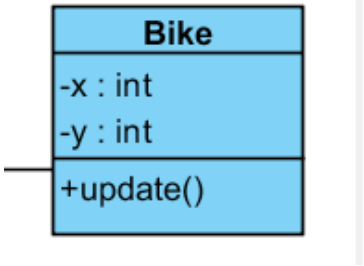
Attributes

private int id: defines kind of enemy.

Constructors

public Enemy(): constructs a proper enemy object according to game.

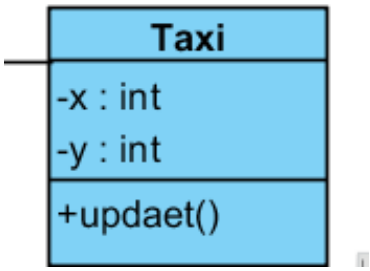
Bike Class



Methods

Public void update(): updates x and y axis values of the bike object according to game

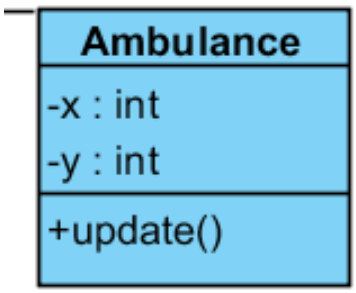
Taxi Class



Methods

Public void update(): updates x and y axis values of the taxi object according to game

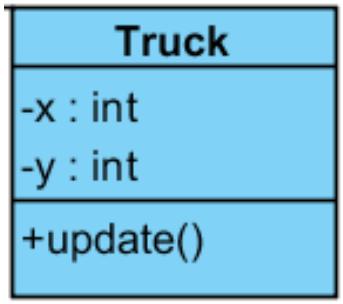
Ambulance Class



Methods

Public void update(): updates x and y axis values of the ambulance object according to game

Truck Class



Methods

Public void update(): updates x and y axis values of the truck object according to game