
Bilkent University

Department of Computer Engineering

# Object Oriented Software Engineering Project

*Crossit!*

## Design Report

Umutcan Aşutlu, Sarp Saatçıoğlu, Figali Taho, Utku Uçkun

Supervisor: Uğur Doğrusöz

Nov 28, 2016

# Table of Contents

# Table of Figures

# 1 Introduction

## 1.1 Purpose of the system

Cross-It is a version of the arcade game Frogger. Our aim is to make the game fun and enjoyable for all people willing to play, therefore the system will be designed such that it makes sense and it is easily grasped by everyone.
The goal of the system is that the player passes as many stages as possible to reach higher and higher scores. A finished stage is  a stage in which the player has crossed the road.

## 1.2 Design goals

### End User Criteria

**Ease of use**

Cross-It will be easy to use so the user need not have previous experience in gaming. The actions the user needs to do are intuitive and straightforward. In case of doubt, the help menu will assist the user in understanding how to play the game better.

**User friendly**

Cross-It is a user friendly game. The setup for the user is very simple, the only thing needed to be previously installed is the Java Runtime Environment.

### Performance Criteria

**Response Time**

The game is an interactive one, so we will make sure the response time will not exceed a certain small threshold, i.e there will be no delay in the game.

**Well defined interfaces**

The project will have a well defined interface, with well defined objects such as player, vehicles such as motorcycle, truck and car. There will also be animations for the collisions, or the different collectibles. The game needs to have good memory usage and need to have enough space to improve performance and speed, java virtual machine (JVM) will delete destroyed objects such as bullets, bugs etc

### Maintenance Criteria

**Reliability**

We plan to create a reliable system. It will not crash or give any errors during runtime. The flow will be smooth, and the user will not experience any unpleasant experience.

**Readability**

The source code will be very readable, so that it is understood by other developers. The source will be self documenting, and the code will look as uniform as possible and with a very consistent style.

**Modifiability**

We will make sure that updates to the system will be possible without much trouble. We, or anyone ready to modify/extend the code, should be able to easily work on the existing system in the future.

# 2 Software Architecture

## 2.1 Subsystem Decomposition

For designing and implementing our system more easily we divided our components into different subsystems. To keep things more consistent we kept components with similar services under the same subsystem. Our system suits the M-V-C (model-view-controller) type of subsystem organization the most.

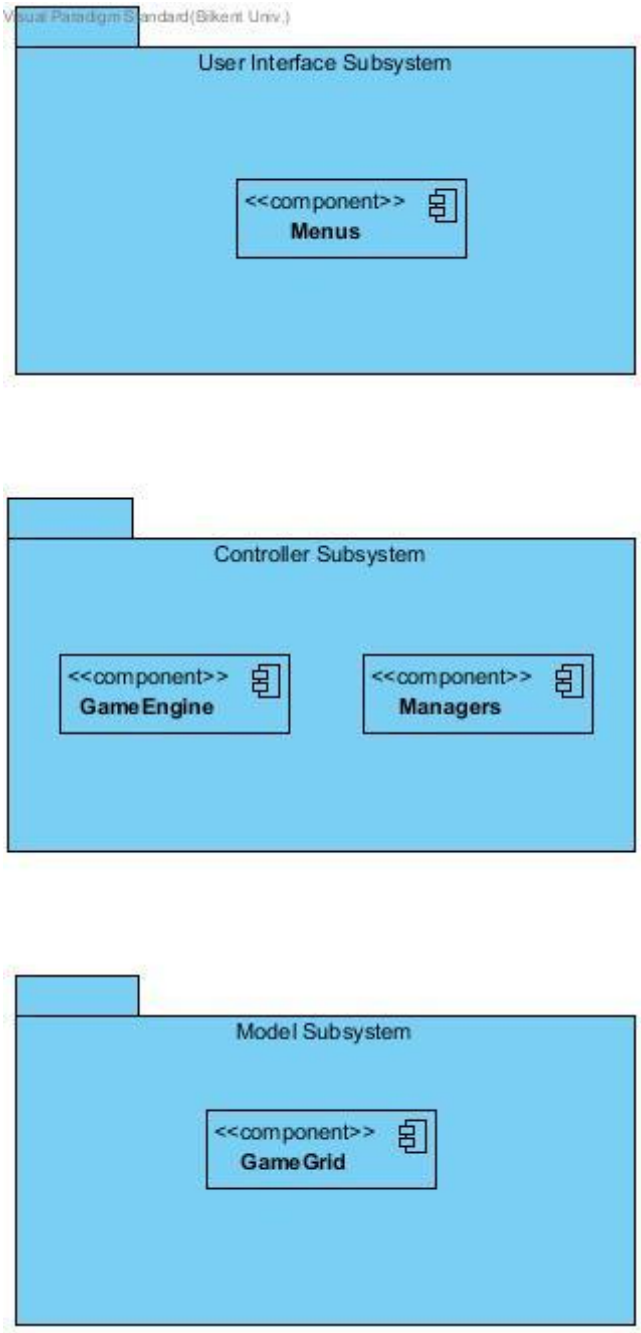


Figure 1:Subsystems of system

### 2.1.1 User Interface Subsystem

User interface subsystem's general purpose is displaying menus in our program. We will use panels to draw these menus on to screen. Game's current state will also be drawn by this subsystem's components

### 2.1.2 Controller Subsystem

Under Controller Subsystem, we grouped all the components that control and manage system's different functionalities. For example, component that manages settings, files and main controller; GameEngine.

### 2.1.3 Model Subsystem

All of our game objects and the grid that keeps these objects is under model subsystem. Model subsystem represents and manages the current state of the game; where cars and player are at in that specific moment etc.

## 2.2 Hardware/Software Mapping

The hardware configuration of the game will consist of keyboard and mouse. The keyboard will be used by the users to enter the input for moving the player (up, down, left or right), and also for entering the name at the End of Game Menu.
For the user to be able to run and play the game the computer needs to have an operating system with Java compiler installed. Other than the programming language, the system needs to also support .png, .txt and .wav files. The .wav files will be for the game sounds, the .txt files will be used for storing the high score list and the settings of the game, and the .png files will be for the interface objects of the game.
It will be an offline game, so there is no need for internet connection.

## 2.3 Persistent Data Management

For our system we have considered using a .txt file for saving the game data, i.e. the settings, player information and scores will be registered. If the file is somehow corrupted, an error will be displayed, and the system will be reset to default. The jar of the game will contain this file as well as the icons and sounds of the game.

## 2.4 Access Control and Security

Cross-It is a singleplayer game that does not require authentication process to start the game. Since there is no authentication process the game will be playable without any internet connection. All the data about the highscores and pre-chosen selections of settings will be stored in a way that user will not be able to change the content of the file of data externally. The game will not have user profiles, therefore there will not be any pre-condition for starting the game.

## 2.5 Boundary Conditions

**Initialization**

- Cross-It starts by an executable file. JVM runs on background and Main menu shows up.
- In the main menu there are five buttons: New Game, Settings, Highscores, Credits, Help, Exit
- When New Game button is clicked, system will create the first stage of the game.
- If player hits the Help button, Help panel will open with the help content.
- The game graphics are made by Adobe Photoshop Cs6 [1].
- If the Settings button is selected, the settings screen will pop-up to customize the game.
- If the Credits button is selected, the general information about game will be displayed, such that version and authors of the game.
- If the Highscores button is selected, highest 10 scores will be displayed with the names on the screen.
- If the Exit button is selected, the game will shut itself down.

**Termination**

- Player can quit the game by pressing the Exit button on the main menu.
- During the game, if player closes the game directly with any method, game will exit without saving any changes such as scores and coins gained from that game session.
- If player wants to exit with the Exit button on the main menu, a prompt that asks whether player is sure or not will be displayed. If player selects yes, the game will quit regularly.

**Failure**

- The game needs to JRE in the system to launch.
- Although the files that stores information i.e. pre-saved settings, highscores are secured, they may be deleted or somehow unreadable. In that situation the game system will assign new default files with deleting previous ones.
- If somehow there occurs an immediate termination, like power cut, current data will be lost.
- If the game terminates unconditionally during a save process, the save process may not be successful.
- If user deletes anything accidentally, an error message pop-up to notify.

# 3 Subsystem Services

The game system will based on three major subsystems. These are Model, View, Controller. The services of these subsystems provides are described in further detail below:

## 3.1 Services of the Model

The model subsystem is composition of all classes of modeling. The model classes consist the main data of the game, like properties and positions of the objects in the game. Any changes that happen during the game session or change in panels will be send to Controller subsystem to process of the game further. This operation of Controller which is gathering information will be supported by sendData service of the Model subsystem. This service will send current locations of the models, player's outfit etc., i.e. anything about the objects of the game that are in the session. The modify call of the controller subsystem also can make changes in Model Subsystem, so there is a bridge between these two subsystems. During the session of the game, the flow of information will be continuous.

## 3.2 Services of the View

The services of the View subsystem provide is everything that user interacts during the session of run. Every element of visualization will be updated accordingly. The first service that will be used is draw. The visual instances of the game will be updated frequently with high-speed, so every change that differs between previous frame will look natural. The update service will be handled by Controller Subsystem with the information that has been provided by Model Subsystem. As an example, every time any vehicle unit or player changes its location, it will be spotted by update service that continuously called, and informs the controller subsystem. Required changes of information will be gathered from Model subsystem to Controller Subsystem. The general composition of this data will be send back to View subsystem's draw services and the panels and frames will be drawn and change accordingly.

## 3.3 Services of the Controller

This subsystem is a set of controller objects that are linking the model and view subsystems. Since this subsystem is responsible to manage all the other subsystems, it has access to all other services. The controller subsystem gets the events from view subsystem and manages the model according to these events. For instance, if the player wants to start a new game by pressing the "New Game" button from the main menu, this event will be handled by the corresponding ActionListener that is provided in related controller object. This controller object modifies the corresponding model object and updates the view with respect to that modification. These kind of handling operations are managed by the controller subsystem with two kind of calls to the other subsystem. First type of call is modify, which notifies model subsystem to process changes passed in the event. Secondly, controller calls the view subsystem to update the view, in our case this is user interface.

# 4 Low-level Design

## 4.1 Object Design Trade-Offs

### 4.1.1 Functionality vs. Ease of Use

In Cross-it we want the player to have an easy and fun time. This is why we decided to keep our game as simple as possible. We didn't try to add flashy, not necessary functions while still keeping our game different than those in the market. We kept the core functions of a Frogger type of games and added some extra spice such as mystery-boxes and purchasables to make our game stand out.

### 4.1.2 Rapid Development vs. Security

Our system uses .txt files to store game information such a user's hats, user's coins, etc. This information should be kept hidden from the user so that he\she cannot manipulate the game as he\she wishes but since we had limited time for implementing our system we decided not to encrypt these files but hide them from the user. A experienced computer user can still access these files but we hope the average computer user cannot find these files.

## 4.2 Final Object Design

We finalize the object diagram as below:

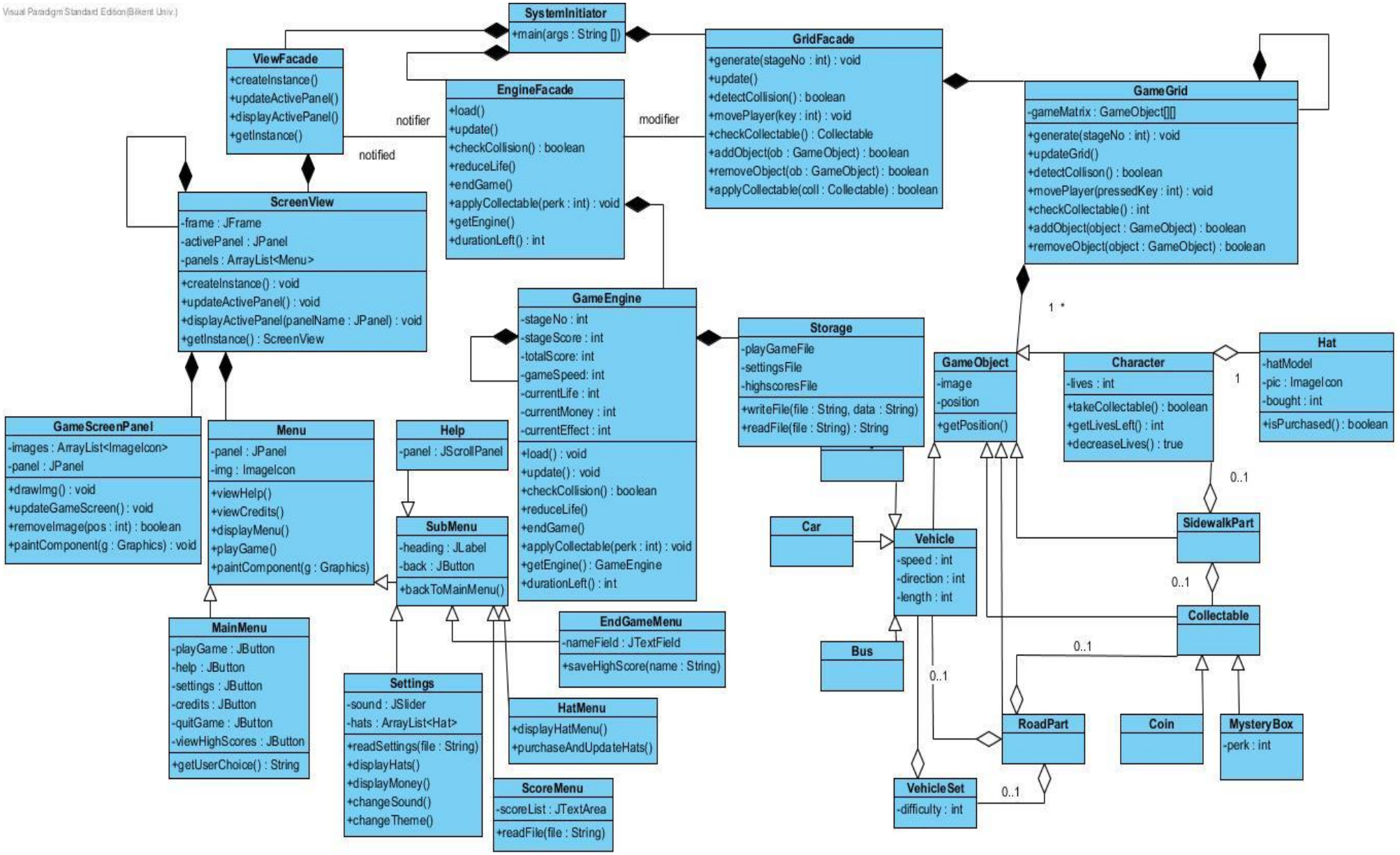


Figure 2: Final Object Diagram

### 4.2.1 Use of Inheritance

In our system, we keep all our entity objects in the same matrix. And these entity objects share some attributes such as a position and an image. Thus, we decided to use an inheritance pattern for these classes and named it GameObject class. GameObject has the shared attributes and operation of the entity object while different entity objects have specialized attributes and operations on their own.

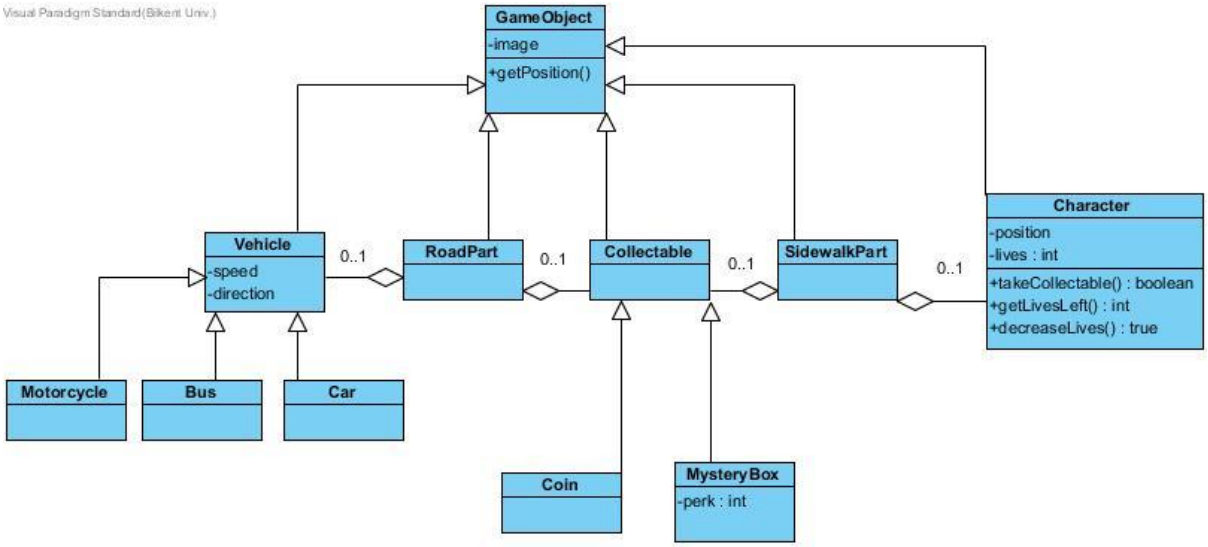


*Figure 3: Use of inheritance*

### 4.2.2 Use of Facade classes

Since one of our design goals is modifiability we decided to go with model-controller-view design in our project. To further serve this purpose we also decided to use facade classes. Each subsystem has a facade class to work like an interface for that subsystem. With this design decision, we make future changes in the subsystems more easy and convenient. With this design when someone changes the internal functioning of a subsystem we do not need to

alter other subsystem as well since they only communicate with the facade class of the altered subsystem.

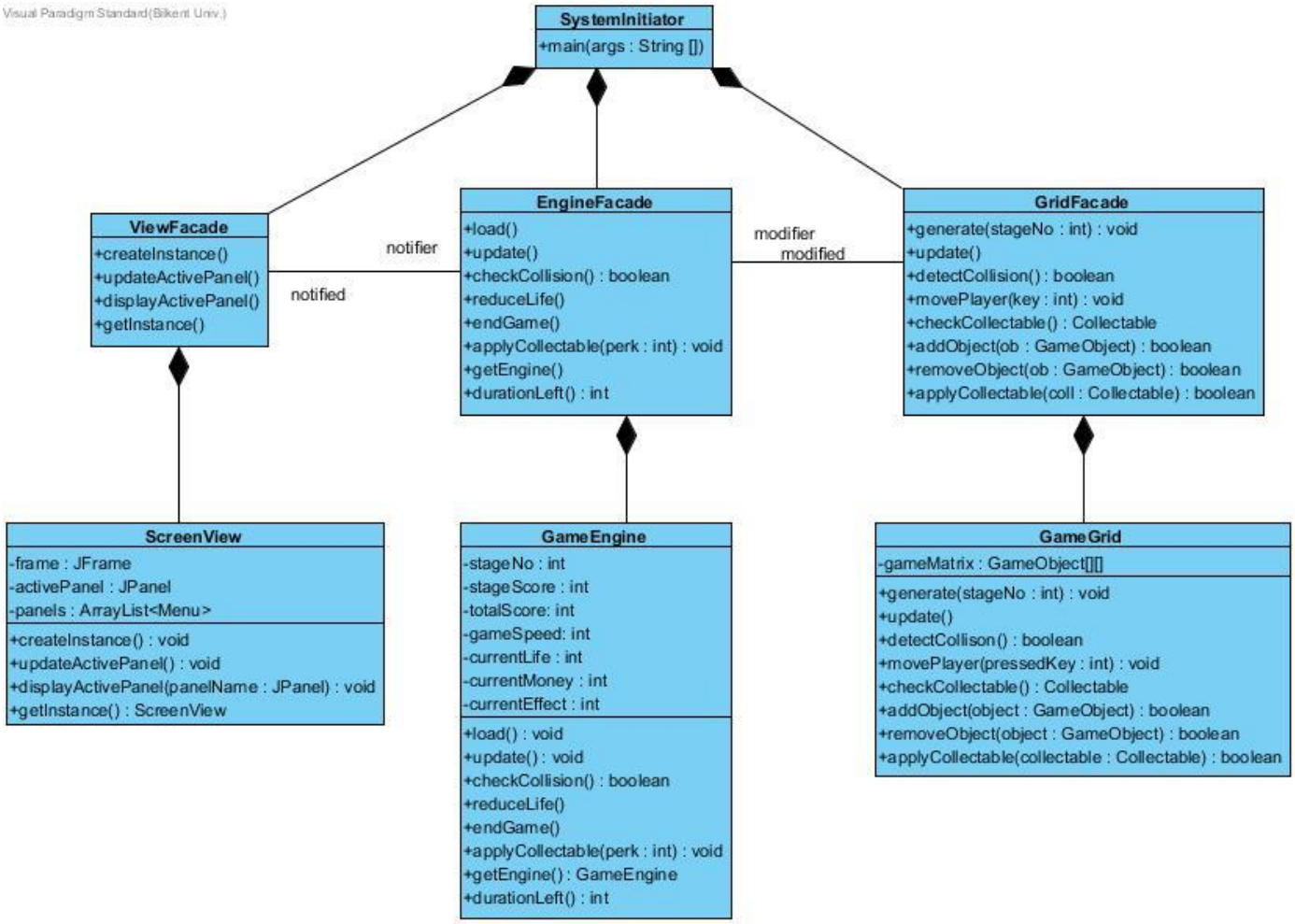


Figure 4: Use of facade classes

### 4.2.3 Use of Singleton

In order to make sure that all our important class in all our subsystems cannot be duplicated we use singleton technique on our system. In order to not have two GameEngines other class can only reach GameEngine with a static method which returns a reference to only GameEngine class. Same methodology goes for ScreenView and GameGrid.

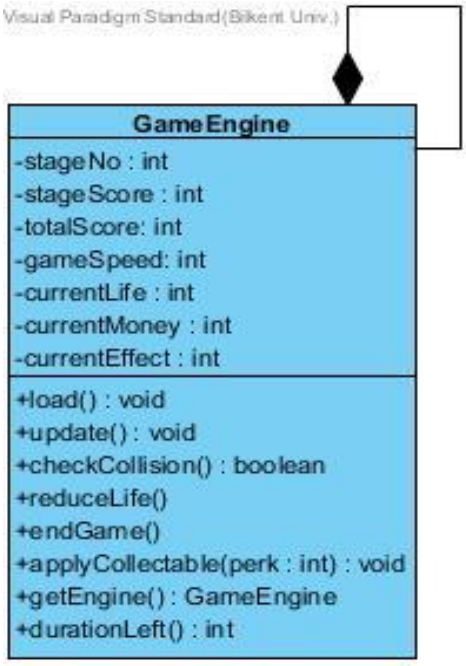


Figure 5:Use of singleton

### 4.2.4 Use of information hiding

Since we want to keep our system reliable and robust we hide unnecessary information from users. If an attribute or an operation is only used by that class we label it as a private, if it is accessible from another class we make it protected.

## 4.3 Packages

This section is going to explain each package diagram according to the final class diagram. Class relationships will be explained for each subsystem, while there is going to be further explanation of how the subsystems work in detail.

### 4.3.1 User Interface Package

The User Interface Package contains all the frame and necessary panels needed for the view of the system. Included here are the menu panels of all the menus that the game contains. Other than that there is the GameScreenPanel which contains all the information of how the game will look like while playing it. The main class in this diagram is the ScreenView class, and the two classes that are associated to it are a general Menu class and the GameScreenPanel class, since conceptually there is a view of the game, and menu views. The Menu can be perceived as sub-menu and main manu, therefore SubMenu and MainMenu classes exist. SubMenu types are Help, EndGameMenu, HatMenu, ScoreMenu, Settings, and Main Menu is the menu that the user first sees when they decide to play Cross-it.
These classes communicate with the controller classes, and especially with GameEngine, since to display some of the data they need to perform checking from the files that Storage takes care of.
The full package diagram is given below:

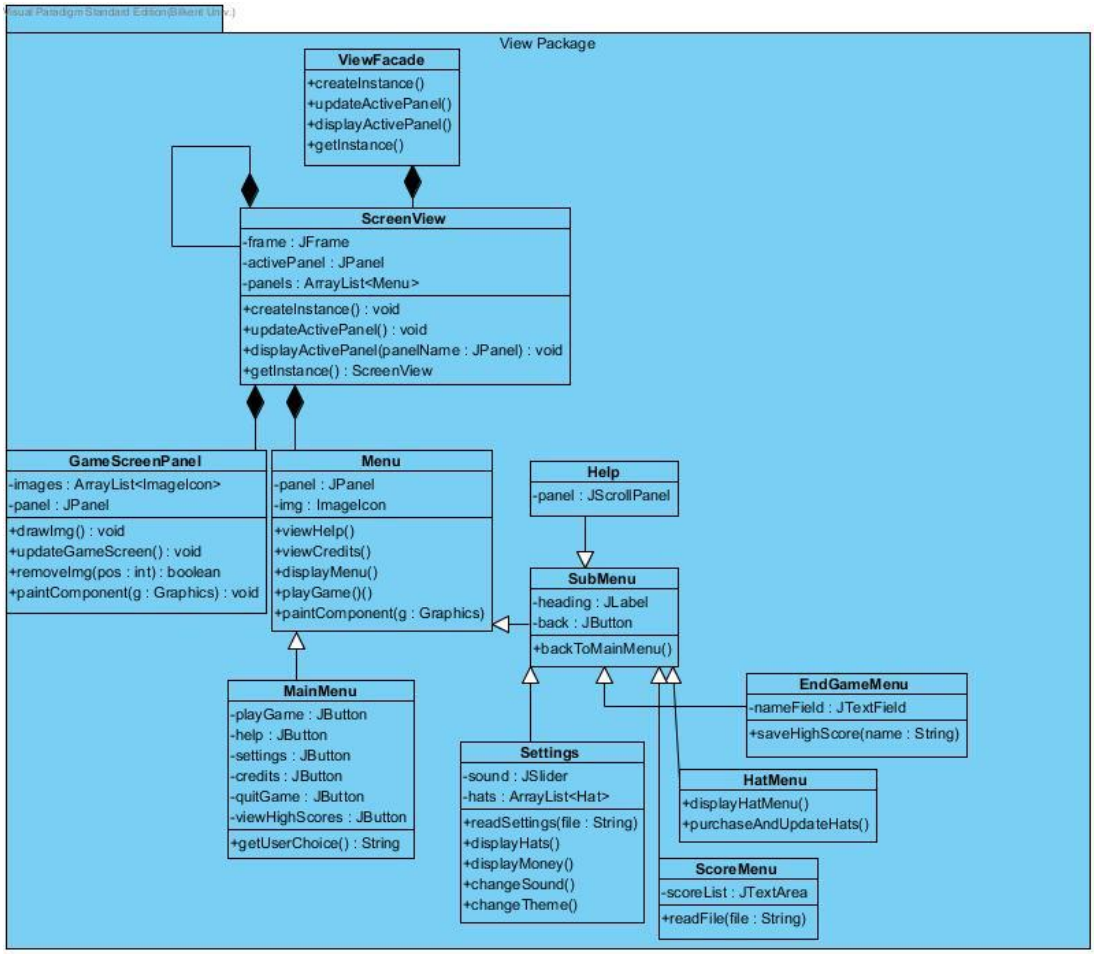


Figure 6:View Package

### 4.3.2 Controller Package

The controller package is the package in which the services of the controller are provided for the system. It is basically composed of a engine or main controller of the game called the gameEngine, and a manager of the Storage. The Storage class takes care of writing to each individual file with which the system communicates to save the data, and reading from files to load the game or specific section belonging to the hats in the settings.
GameEngine is the brains of the system, and makes sure everything is working fine in the game. It communicates with the view classes to make sure updates happen accordingly while the game is going on, and makes sure that the game loads smoothly, the menus indicate correct information and updated accordingly from the storage. Detailed description of each class is provided in the upcoming sections, while the diagram for the package is given right below:

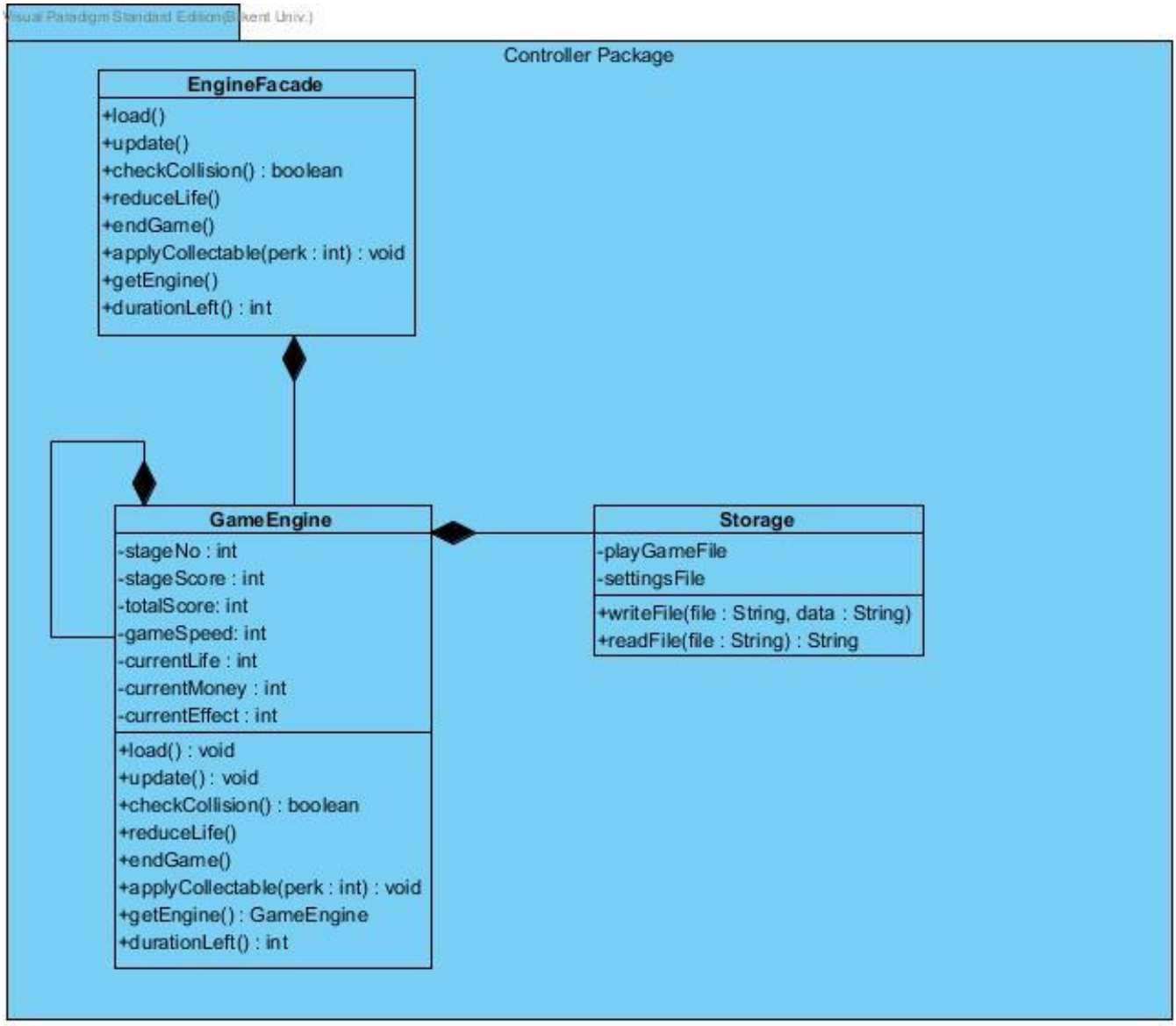


Figure 7:Controller Package

### 4.3.3 Model Package

The model package contains the entity objects of the system. The package is composed of the GameGrid class, which is the class that defines the grid of the game. The GameGrid class is the way the controller subsystem communicates with the model subsystem. The GameGrid holds all the objects of the game, which are depicted as the superclass of GameObject. GameObject objects can be Vehicle, Collectable, RoadPart or SidewalkPart. Each may or may not be composed of other entity objects as in the diagram below:

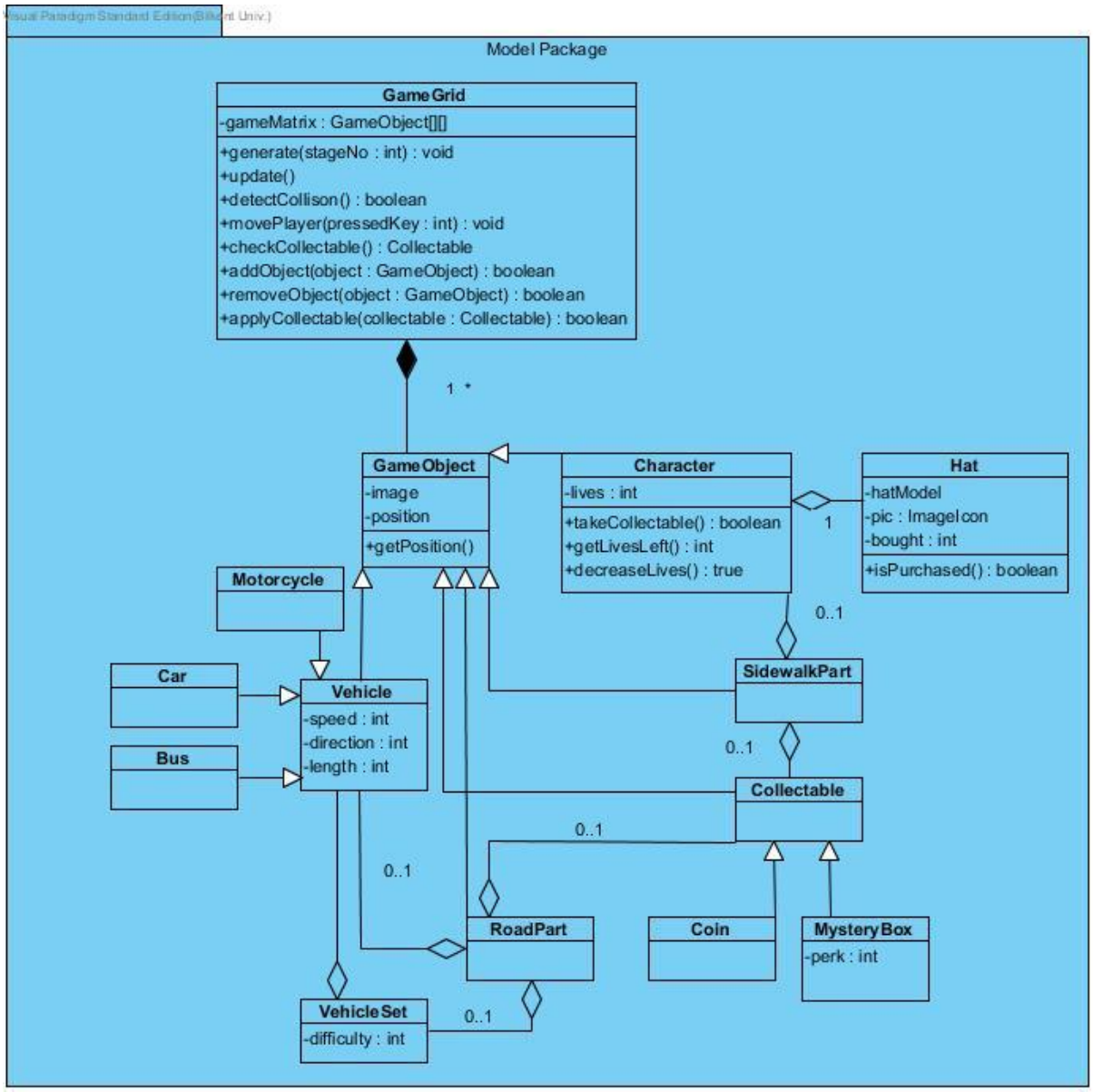


Figure 8:Model Package

# 4.4 Class Interfaces

## 4.4.1 Model Classes

### GameGrid Class

GameGrid class is basically substructure of the game. Everything behind the interface will be represented in here on a two dimensional matrix. Every change that happens here will be occur in interface after on.

- *Constructor :*

**protected GameGrid():** This constructor creates a grid of the game which is simply the whole game mechanic.

- *Attributes :*

**private GameObject[][] gameMatrix:** This attribute is the base of the game grid that is substructure of the game.

- *Methods :*

**protected void generate(int stageNo):** This method generates the stage map by its number.

**protected void updateGrid():** This method will constantly update grid according to changes happened in GameEngine class.

**protected boolean detectCollision():** This method detects any collisions between player and any entity object.

**protected void movePlayer (int pressedKey):** This method changes player position by the given key as parameter.

**protected Collectable checkCollectable():** This method checks the type of the collectable.

**protected boolean addObject(GameObject object):** This method adds an object to grid.

**protected boolean removeObject(GameObject object):** This method removes an object from grid.

## GameObject Class

This class is general class for the all objects in the game since they have some common attributes. The position and image of the objects will be held in this class.

- *Constructor :*

**protected GameObject(ImageIcon image, int x, int y):** The game objects will be created and initialized in this constructor.

- *Attributes :*

**private ImageIcon image:** This attribute is the image of the object which is given by parameter.

**private int positionX:** This attribute declares the position of the game object on X coordinate.

**private int positionY:** This attribute declares the position of the game object on Y coordinate.

- *Methods :*

**protected int getPositionX():** This method returns the position value of the game object on X coordinate.

**protected int getPositionY():** This method returns the position value of the game object on Y coordinate.

## Character Class

Character class is for creating a model which can player can play with. It will be saved directly, however things like such as coins and purchased hats will be. This class is child class of the GameObject class. It uses GameObject class's attributes as well as itselfs.

- *Constructor :*

**protected Character():** The player object will be created and initialized in this constructor.

- *Attributes :*

**private int lives:** The number of lives is shown in this attribute. The number is setted to be 3 by default value.

- *Methods :*

**protected boolean decreaseLives():** This method will decrease the amount of lives that character has by one.

**protected int getLivesLeft():** Returns the remaining number of lives of the character.

**protected boolean takeCollectable():** This method will help to pick-up a collectable that exists.

## Hat Class

This class is for hat objects that are in the game which player can buy with coins. Once they purchased, it will remain until somehow user deletes the storage files of the game.

- *Constructor :*

**protected Hat():** The hat object be created and initialized in this constructor.

- *Attributes :*

**private int hatModel:** This attribute is for differentiating the hat from others. Every hat has it's unique hatModel value.

**private imageIcon pic:** This attribute is simply graphical illustration of the hat objects.

**private int bought:** This attribute holds 1 or 0 to give information about if the hat is already purchased or not. It is 0 by default.

- *Methods :*

**protected boolean isPurchased():** This method returns the information of whether the hat is purchased before or not.

## Collectable Class

This class is main class for collectable objects in the game. It is child class of the GameObject class.

- *Constructor :*

**protected Collectable():** The collectible objects will be created by this constructor. It has not any attributes or methods. It is just default constructor.

## Coin Class

This class is child class of the Collectable class. It represents the coins in the game. The amount of coins will be saved with storage class to being able to use later.

- *Constructor :*

**protected Coin():** This constructor is a default constructor that identifies the coin objects in the game.

## MysteryBox Class

This class is child class of the Collectable class. It represents the mystery boxes in the game. It contains random integer number to resemble the effect that it has.

- *Constructor :*

**protected MysteryBoxes():** This constructor is for initializing the mystery box objects in the game.

- *Attributes :*

**private int perk:** This attribute is for identifying the type of the effect that mystery box contains. It will send the information to the game engine and engine will apply the effect corresponds to this attribute's value.

## VehicleSet Class

This class is for creating a vehicle sets by difficulty level which is related to stage number. They will contain a random amount of number according to it's difficulty value. This sets will be assigned to roadParts directly.

- *Constructor :*

**protected VehicleSet(int difficulty):** The constructor for creating and initializing the vehicle sets of the game.

- *Attributes :*

**private int difficulty:** This attribute is for declaring the difficulty level of the game. According to this value the sets will have corresponding number of vehicle and type.

## Vehicle Class

Vehicle class is for all common attributes of the vehicles in the game. They will not be directly linked to the roads. They will be created and used in vehicle sets. This class is child class of the GameObject class. It uses GameObject class's attributes as well as itselfs.

- *Constructor :*

**protected Vehicle(int length, int direction):** The vehicle object will be created and initialized by this constructor with using certain parameters  such as length and direction.

- *Attributes :*

**private int length:** This attribute declares the length of the vehicle. Which also decides the type of the vehicle, since every vehicle has it's own length.

**private int speed:** This attribute declares the speed of the vehicle.

**private int direction:** This attribute sets the direction of the vehicle either 0 or 1 which resembles left and right.

## Motorcycle Class

This class is child class of the Vehicle class. It contains all of the attributes Vehicle class have. Because of it has it's own image, this class should be specified. The length of motorcycle  class objects are "1" units.

- *Constructor :*

**protected Motorcycle(int direction):** The motorcycle object will be created and initialized by this constructor.

## Car Class

This class is child class of the Vehicle class. It contains all of the attributes Vehicle class have. Because of it has it's own image, this class should be specified. The length of car class objects are "2" units.

- *Constructor :*

**protected Car(int direction):** The car object will be created and initialized by this constructor.

## Bus Class

This class is child class of the Vehicle class. It contains all of the attributes Vehicle class have. Because of it has it's own image, this class should be specified. The length of bus class objects are "3" units.

- *Constructor :*

**protected Bus(int direction):** The bus object will be created and initialized by this constructor.

## SidewalkPart Class

This class is for identifying the road that have no any vehicle on it. However, there can be Collectable objects on this type of road.

- *Constructor :*

**protected SidewalkPart():** The sidewalk object will be created and initialized by this constructor.

## RoadPart Class

This class is for identifying the road that have vehicles on it. As well as vehicles, there can be Collectable objects on this type of road too.

- *Constructor :*

**protected RoadPart():** The road object will be created and initialized by this constructor.

### 4.4.2 View Classes

## ScreenView Class

This class constructs the general view of the game.

- *Constructor :*

**protected ScreenView():** This method will do nothing except initializing the object.

- *Attributes :*

**private JFrame frame:** The frame of the screen is initialized in this attribute.

**private JPanel activePanel:** The panel that is initialized after the frame attribute.

**private ArrayList<Menu> panels:** This attribute holds an ArrayList of Menu objects.

- *Methods :*

**protected void createInstances():** This method will initialize the ScreenView object with it's attributes.

**protected void updateActivePanel():** Updates the activePanel view.

**protected void displayActivePanel(JPanel panel):** This method will display the specified panel on the SacreenView object.

**protected ScreenView getInstance():** This method returns the current object. It is used as a regular getter method.

## GameScreenPanel Class

GameScreenPanel displays the game as a view and it listens the actions of the player in order to update itself.

- *Constructor :*

**protected GameScreenPanel():** This method will initialize the object.

- *Attributes :*

**private JPanel panel:** The panel that contains visual elements.

**private ArrayList<ImageIcon> images:** This attribute holds the external visual elements of the game screen.

- *Methods :*

**protected void drawing():** This method will display the view.

**protected void updateGameScreen():** This method will update the view of game screen.

**protected boolean removeImg(int position):** This method will remove the specified imageicon object from the game screen view.

**protected void paintComponent(Graphics g):** This method is a generic Java method for Graphical User Interface implementation. It sets up the visual configurations.

## Menu Class

Menu class provides the basics of the Menu view of the game and provides functionality to the menu elements.

- *Constructor :*

**protected Menu():** This method will initialize the Menu object.

- *Attributes :*

**private JPanel panel:** The panel that contains other attributes.

**private ImageIcon img:** This attribute holds the icon of the game.

- *Methods :*

**protected void viewHelp():** This method will display the help view.

**protected void viewCredits():** This method will display the credits view.

**protected void displayMenu():** This method will display the menu that is initialized as this object.

**protected void playGame():** This method sets up the game.

**protected void paintComponent(Graphics g):** This method is a generic Java method for Graphical User Interface implementation. It sets up the visual configurations.

## Help Class

This provides a view to display the help content to the player.

- *Constructor :*

**protected Help():** This method will initialize the Help object.

- *Attributes :*

**private JScrollPane panel:** The panel that contains help data.

## MainMenu Class

MainMenu class is to view the options of the game and choose what player wants to do with the game.

- *Constructor :*

**protected MainMenu():** This method will initialize the MainMenu object.

- *Attributes :*

**private JButton playGame:** The button that listens user choice.

**private JButton help:** The button that listens user choice.

**private JButton settings:** The button that listens user choice.

**private JButton credits:** The button that listens user choice.

**private JButton quitGame:** The button that listens user choice.

**private JButton viewHighScores:** The button that listens user choice.

- *Methods :*

**protected String getUserChoice():** This method will return the choice of the user in terms of clicked button.

## Settings Class

Settings class provides a view of settings that holds settings data from a file and allows player to change the settings.

- *Constructor :*

**protected Settings():** This method will initialize the Settings object.

- *Attributes :*

**private JSlider sound:** This attribute provides a slider to control sound volume of the game.

**private ArrayList<Hat> hats:** This attribute holds the Hat objects that are avaişlable to purchase.

- *Methods :*

**protected void readSettings(String file):** This method takes the name of the file that contaiğns data of settings and sets up the sttings from that data.

**protected void displayHats():** This method will display the hats in the settings view.

**protected void displayMoney():** This method will display money that player earned that is taken from settings file.

**protected void changeSound():** This method changes the sound with respect to the data in the JSlider object of the view.

**protected void changeTheme():** This method changes the theme of the game.

## HatMenu Class

The HatMenu class allows player to purchase and change the Hat that is displayed in this view.

- *Constructor :*

**protected HatMenu():** This method will initialize the HatMenu object.

- *Methods :*

**protected void displayHatMenu():** This method will display the HatMenu view.

**protected void purchaseAndUpdateHats():** This method updates the hat of the player with respect to the purchase choice in the HatMenu view.

## ScoreMenu Class

This class provides a view that allows player to see the recorded scores that are stored in a file.

- *Constructor :*

**protected ScoreMenu():** This method will initialize the ScoreMenu object.

- *Attributes :*

**private JTextArea scoreList:** This attribute holds the score that user scored in the current game session.

- *Methods :*

**protected void readFile(String file):** This method takes the name of the file that contains data of scores.

## EndGameMenu Class

This EndGameMenu class is the view that occurs when the game ends and it provides player to record his/her score to the recorded scores file.

- *Constructor :*

**protected EndGameMenu():** This method will initialize the EndGameMenu object.

- *Attributes :*

**private JTextField nameField:** This attribute takes the name of player in order to use for recording score.

- *Methods :*

**protected void saveHighScore(String name):** This method takes the name of the player that is hold in the nameField object and records the score to the file that stores data of the scores.

## SubMenu Class

This SubMenu class is the super class of the Menu classes.

- *Constructor :*

**protected SubMenu():** This method will initialize the SubMenu object.

- *Attributes :*

**private JLabel heading:** This attribute holds the header as a visual object.

**private JButton back:** This attribute is a button that listens the action of the player to go back to main menu.

- *Methods :*

**protected void backToMainMenu():** This method will be called from the action listener of the back JButton object and goes back to the main menu.

### 4.4.3 Controller Classes

## GameEngine Class

The main functionality of the game will be done by the GameEngine. It is the main controller of the game and updates the game objects according to it's own functions and user's inputs.

- *Constructor :*

**private GameEngine():** The constructor of the engine of the game that will set all of the attributes.

- *Attributes :*

**private int stageNo:** This attribute holds the current stage number of the game.

**private int stageScore:** This attribute holds the score that has achieved in current stage.

**private int totalScore:** This attribute holds the score that has achieved from the beginning of the game session.

**private int gameSpeed:** This attribute is indicates the speed of the game which is the component of initializing the speed of the vehicles.

**private int currentLife:** This attribute demonstrates the left lives of the character.

**private int currentMoney:** This attribute demonstrates the money of the character that has been collected from very first session in game.

**private int currentEffect:** This attribute demonstrates the effect that have been applied by mystery boxes, if there is any. It will be set 0 by default which means no effect.

- *Methods :*

**protected void load():** This method load the content that have been used previous session such as money has been collected and applied settings.

**protected void  update():** This method will constantly update the game, the update is about applying changes that requested.

**protected boolean checkCollision():** This method checks if there are any collisions between player and any entity object.

**protected void reduceLife():** This method will decrease the total lives of the character by one if there are any collision beforehand.

**protected void endGame():** This method ends the current session of the game and send player to the main menu.

**protected void applyCollectable(int perk):** This method will apply the effect that has been given in parameter to the game.

**protected gameEngine getGameEngine():** Regular getter method for the game engine.

**protected int durationLeft():** This method works like a timer that demonstrates the left duration of the effect that has applied.

## Storage Class

Storage class's duty is reading the the file which includes some content that saved previous session of the game and writing the new content from game to the file. It has 2 files that hold content.

- *Constructor :*

**private Storage(String playGameFile, String settingsFile, String highscoresFile):** This constructor is default constructor which gives the file names to initialize the attributes.

- *Attributes :*

**private playGameFile:** This attribute is responsible for initial configuration of the matrix.

**private settingsFile:** This attribute is responsible from settings datas such as theme and volume.

**private highscoresFile:** This attribute is responsible from game content datas such as highscores.

- *Methods :*

**protected writeFile(String fileName, String Data):** This method writes the given data to given name of file to store the information.

**protected String readFile(String fileName):** This method reads the data from given name of file and returns the data.

# 5 References

[1] A. S. Incorporated, "Photoshop: For windows: Adobe Photoshop 13.0.1.3 update for CS6:Thank you," 2016. [Online]. Available: https://www.adobe.com/support/downloads/thankyou.jsp?ftpID=5677&fileID=5701. [Accessed: Nov. 12, 2016].