

# Software Engineering Course

## C# Project

### Parking Lot Cashier Software

2013/2014 Summer Session

*Student:*

**Tutu Isaac Asamoah (246162)**

*Professor:*

**Edoardo Bontà**

## PROBLEM SPECIFICATION

Parking garages often have complicated billing procedures. The people who own the garages like to have a computer figure out how much money is owed so that the people working there do not have to waste a lot of time thinking when people are leaving. The project is to write a computer program that collects some simple pieces of information and tells the cashier how much money to collect.

## REQUIREMENTS ANALYSIS

### Overview

Write a computer program that collects some simple pieces of information and tells the cashier how much money to collect. We will use the time function to have the program figure out how much time has been spent at the parking lot.

Charge is based on two forms of parking:

1. **Short term:** frequent use of the parking lot, and can be

- Standard parking
- Event-based parking

1. **Long term:** monthly or yearly

We will keep track of the duration for which cars are parked. Considering long term parking, a database will be created to store client information to keep track of their records. Unique system code will be assigned to each client which will be used to query the database to verify the client's booking information.

For short term parking periods, we make sure that normal charges are made within two days of parking. If a car is parked for more than two days, a notice is issued and a constant fine is added to the normal fee.

## **Software Context Functionality**

In order to resolve the issue, we must change the way that the staff at the parking lot gather necessary information both for client registration and fee generation.

The system will be implemented in order to take care of these issues technically, in such a way that; if for longer period parking, the user is registered on the system with the necessary information.

At the time of exit the system uses the user code to retrieve all the information for that user. By this the system automatically generates the needed fee.

Initially the system asks if we want to perform operation on customer or we just want to generate a due fee.

Generating a fee entails the following steps:

1. get the entry time
2. system automatically generates total time used using the current system time
3. choose the type of car
4. select the type of payment, either standard type or event type
5. an option to give out discount

If we want to perform operation on a client

## **Data Description**

### **inputs**

- entry time – input from the user
- exit time – input from the system
- type of vehicle – input from the user
- type of payment - input from the user
- discount - input from the user

#### **outputs**

- fees
- client data (id, name, entry time, parking type, type of car)

## **Usage scenario**

### **User profiles**

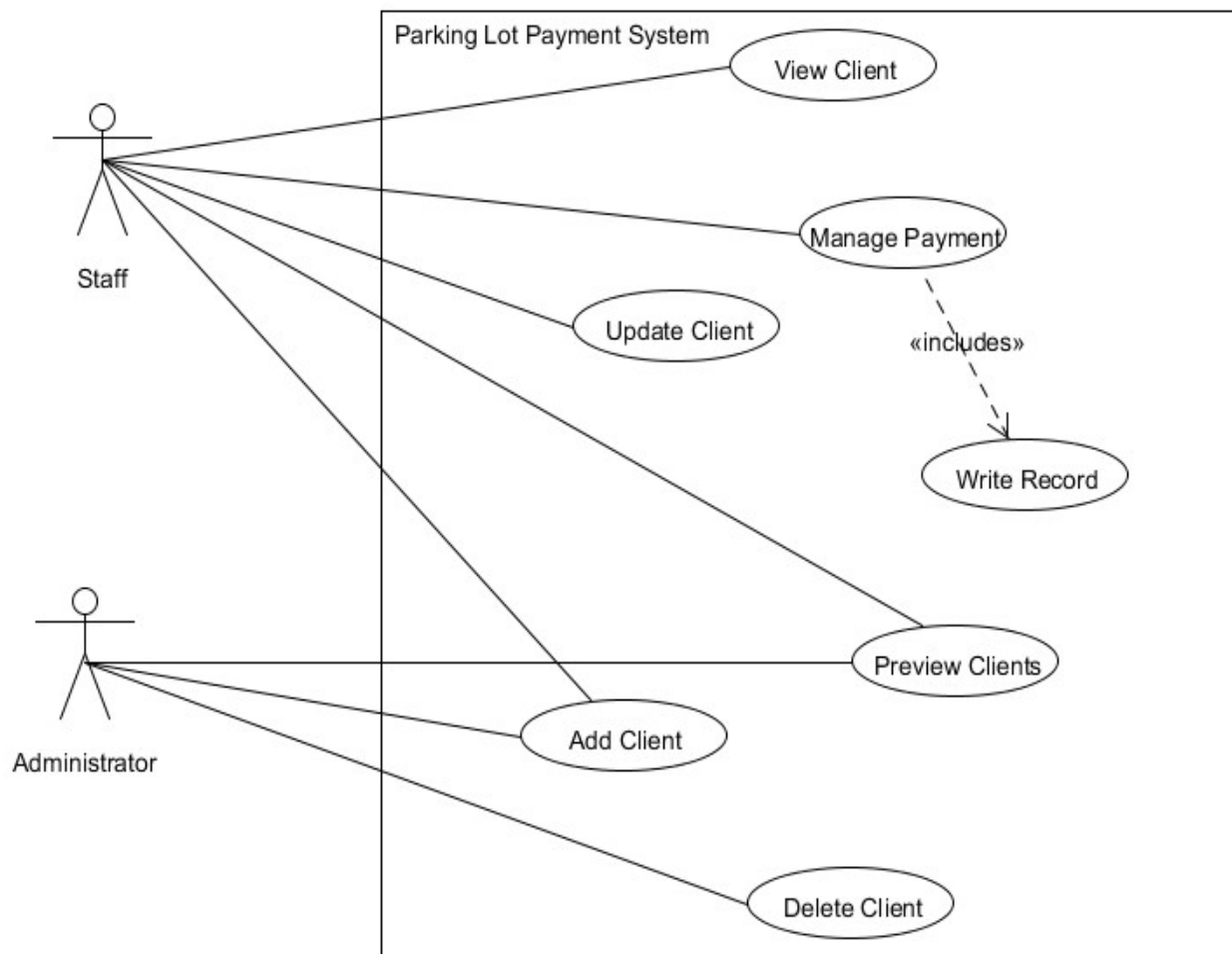
*Staff* – these individuals oversee the parking garage and all of its aspect. They are responsible for maintaining prices, client issues such as payment problems, login issues, and general parking garage problems.

*Administrators* – these individuals perform the administartive tasks and some client operations.

## **Use cases**

Although the Parking Lot Payment system is a relatively complex system from the theoritical point of view, its direct interaction with the user is relatively simple. The primary actors in this system are *administrators* and *staff*. Administrators use the system to manage payments, update, view and add or users. The *administrator* is solely responsible for deleting a client from the system.

## Use Cases Diagram



## Use Cases Specifications

<b>Use case:</b> Manage Payment
<b>ID:</b> UC1
<b>Actor:</b> Staff
<b>Preconditions:</b> <ul style="list-style-type: none"><li>• Staff is at main prompt</li><li>• Staff is presented with a choice to select an option</li></ul>
<b>Basic course of events:</b> <ol style="list-style-type: none"><li>1. Staff selects manage payment option</li><li>2. System asks for entry time</li><li>3. Staff inputs entry time</li><li>4. System asks for type of vehicle</li><li>5. Staff selects type of vehicle</li><li>6. System asks for type of payment</li><li>7. Staff selects type of payment</li><li>8. System asks if discount is available</li></ol>

9. Staff inputs a yes/no answer
10. System asks for discount
11. Staff inputs discount
12. System displays fee

**Postconditions:**

- The required fee is generated and displayed to Staff

**Alternative paths:**

At 9, if Staff inputs no, the system generates the fee without discount

**Use case:** Add Client

**ID:** UC2

**Actor:** Administrator, Staff

**Preconditions:**

- Administrator/Staff is at the manage client prompt
- Administrator/Staff selects add client option

**Basic course of events:**

1. Administrator/Staff inserts client information into the database
2. Information is verified in database
3. Client is added in the database

**Postconditions:**

- Client is registered on the system

**Alternative paths:**

None

**Use case:** Delete Client

**ID:** UC3

**Actor:** Administrator

**Preconditions:**

- Administrator is at the manage client prompt
- Administrator selects delete client option

**Basic course of events:**

1. Administrator inserts client's code
2. Code is verified in database
3. Client with the code is deleted

**Postconditions:**

- Client is deleted from the system

**Alternative paths:**

If 2 fails, sytem displays error

<b>Use case:</b> Update Client
<b>ID:</b> UC4
<b>Actor:</b> Staff
<b>Preconditions:</b> <ul style="list-style-type: none"> <li>• Staff is at the manage client prompt</li> <li>• Staff selects update client option</li> </ul>
<b>Basic course of events:</b> <ol style="list-style-type: none"> <li>1. Staff inserts client code along with client information to be updated</li> <li>2. Code and information are verified in database</li> <li>3. Client information is updated in database</li> </ol>
<b>Postconditions:</b> <ul style="list-style-type: none"> <li>• Client is updated on the system</li> </ul>
<b>Alternative paths:</b> <p>If 2 fails, system displays error</p>

<b>Use case:</b> View Client
<b>ID:</b> UC5
<b>Actor:</b> Staff
<b>Preconditions:</b> <ul style="list-style-type: none"> <li>• Staff is at the manage client prompt</li> <li>• Staff selects view client option</li> </ul>
<b>Basic course of events:</b> <ol style="list-style-type: none"> <li>1. System asks for client code</li> <li>2. Staff inserts client code</li> <li>3. Code is verified in database</li> <li>4. Client information is displayed to staff</li> </ol>
<b>Postconditions:</b> <ul style="list-style-type: none"> <li>• Clients information is displayed</li> </ul>
<b>Alternative paths:</b> <p>If 3 fails, system displays error</p>

<b>Use case:</b> Preview Clients
<b>ID:</b> UC6
<b>Actor:</b> Administrator/Staff
<b>Preconditions:</b> <ul style="list-style-type: none"> <li>• Administrator/Staff is at the manage client prompt</li> <li>• Administrator/Staff selects preview clients option</li> </ul>
<b>Basic course of events:</b> <ol style="list-style-type: none"> <li>1. System displays all the client information stored in the database</li> </ol>

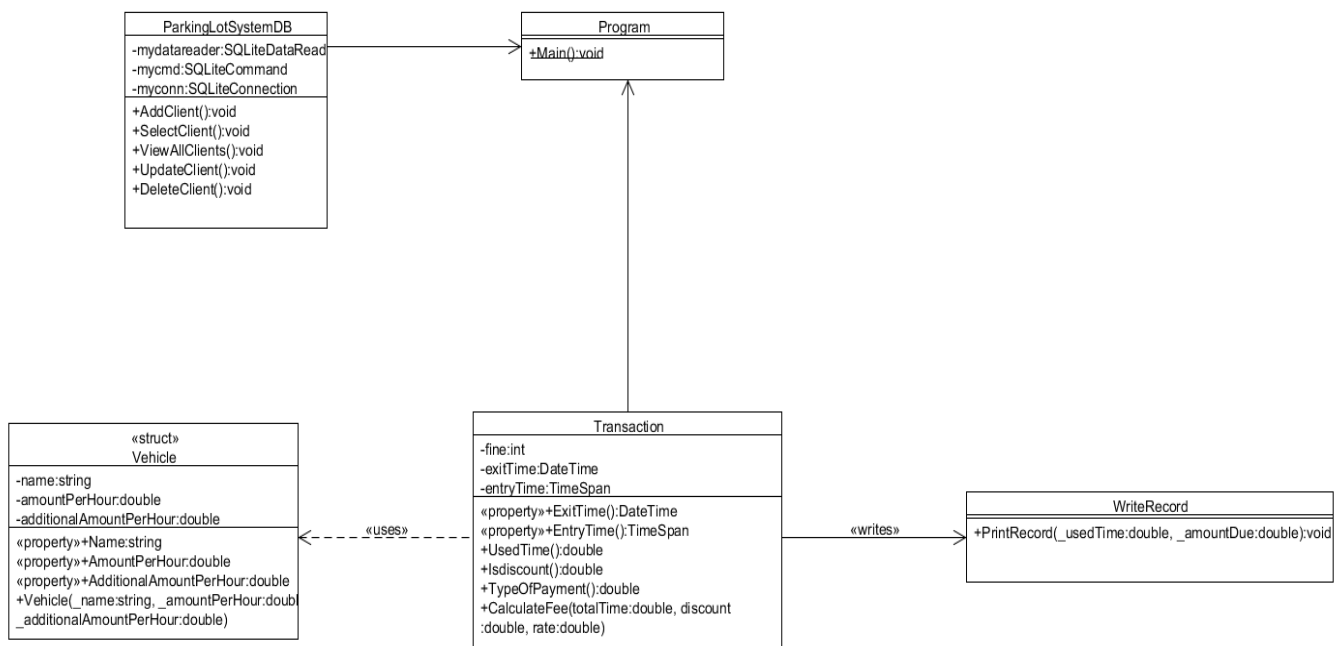
<b>Postconditions:</b> <ul style="list-style-type: none"> <li>• All clients information in the system are displayed</li> </ul>
<b>Alternative paths:</b> None

<b>Use case:</b> Write Record
<b>ID:</b> UC7
<b>Actor:</b> UC1 (Manage Payment)
<b>Preconditions:</b> <ul style="list-style-type: none"> <li>• Record file not updated</li> </ul>
<b>Basic course of events:</b> <ol style="list-style-type: none"> <li>1. UC1 (Manage Payment) writes the current date, entry time, exit time and fee to the file</li> </ol>
<b>Postconditions:</b> <ul style="list-style-type: none"> <li>• Record file updated</li> </ul>
<b>Alternative paths:</b> None

# ANALYSIS AND DESIGN

## Class Diagram

The class diagram for the system together with the description of each class is reported below:



First of all, I have implemented this program using a simple console application (most commonly called C# Console Application) which is made available by the program Microsoft Visual C # 2010 Express, by using clauses that specify what are the namespace in which to look for classes that will be used.

In my case, the program does not use graphical interfaces, however I did not have to worry much about the management of visual components for input/output, just a command line of which I dealt with the decoupling from the functional logic (or model) of the application, using the MVC architectural (design) pattern.

The project was created using a single namespace *ParkingLotSystem*. In the various classes of the project, were interconnected the various classes using the *using* keyword . The project, therefore, is launched from a main class called *Program*. After clarifying the



structure upon which the project is used by the fact that a good software must be extensible and reusable to others, then I tried to build the application providing it with a variety of methods, trying to facilitate them to reuse or to their increasing complexity. Even as the program is very small in nature, I tried to apply to the project the concept of encapsulation ie the division of a complex system in modular units each with a clearly defined functionality.

The code has been written in such a way as to be well-read and understand, well-indented, spaced, and commented on its complexity, in order to make it as clear as possible.

During the design phase, I proceeded in this way: I first implemented the main class of the application before using UML diagrams and then implemented in C#. Thereafter I implement the individual helper classes as if they were independent programs on their own. This venue has been chosen to present the classes individually, in the order in which they were actually implemented using the UML class diagram, viewing the properties of classes such as attributes, and then analyzing the way in which these classes interact, using the UML class diagram and viewing the properties of classes such as associations, so as to have an overall view of how the program operates. A proper error interception/handling strategy will be implemented inside methods.

The program required a database, therefore SQLite was used, as the DB enables application portability across operating systems and does not require activation management systems active and separated from the application (such as MySQL or other).

Following is a brief description of the individual classes along with some considerations and design choices:

### **Class Program**

This is the main class of the program and is the class to start the system. It Instantiates the operation that we want to perform. It contains the *Main(string[] args)* method whose main function is to collect inputs and produce outputs, with a little bit of logic.

### **Class Transaction**

This is the class that performs most of the logics for the fee calculation part of the system.

It Instantiates the *Vehicle* structure object that contains data that it needs to complete its operation.

The class contains the methods *EntryTime* and *ExitTime* which are used to access its private properties *entryTime* and *exitTime* while the private variable *fine* is only used locally within this class. It also contains the operational methods *UsedTime()*, *Isdiscount()*, *TypeOfPayment* and *CalculateFee(double totalTime, double discount, double rate)* which are used to perform its operations. All of the methods are declared as public so that they can be accessed from main from outside the class.

## Struct Vehicle

This is a structure that represents the vehicle object. I chose to use a structure to represent this object because it does not perform any operation, it only contain data that other classes may need for their operations. Therefore, I chose to declare all the properties as private, providing access to them through public methods. It contains *name*, *amountPerHour* and *additionalAmountPerHour* as its private properties and they are accessed through the public methods *Name*, *AmountPerHour* and *AdditionalAmountPerHour*. It also has a public constructor *Vehicle(string \_name, double \_amountPerHour, double \_additionalAmountPerHour)*.

## Class WriteRecord

This is a simple class intended to be used to record the transactions. It writes its output to an external file in the system. It contains only one public method *PrintRecord(double \_usedTime, double amountDue)* which is used to record the date of the transaction, the total time used and the fee calculated.

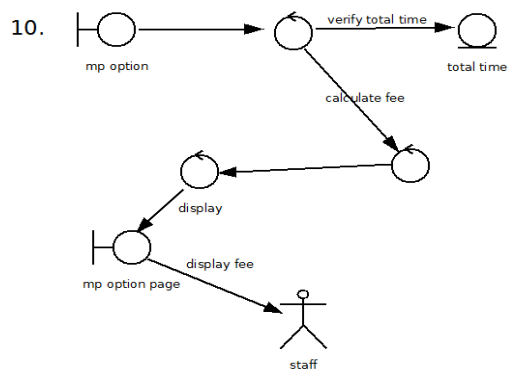
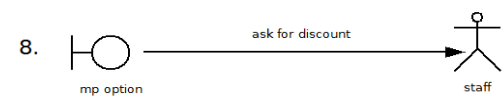
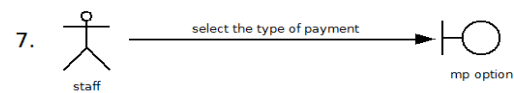
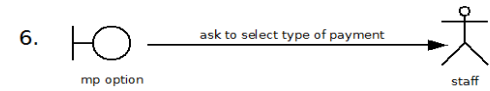
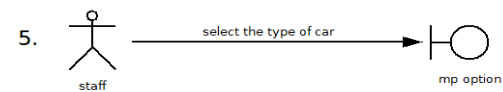
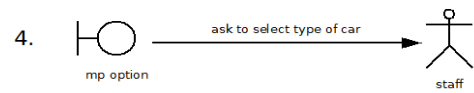
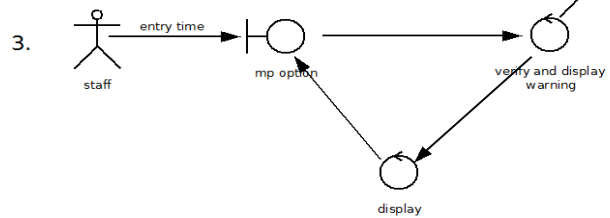
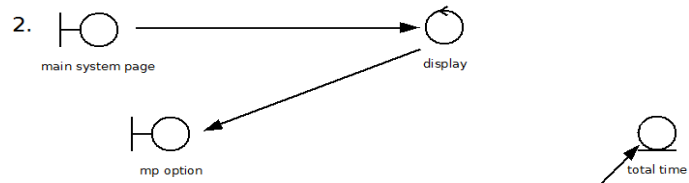
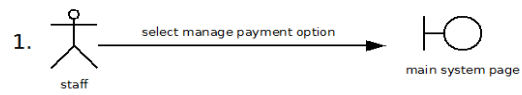
## Class ParkingLotSystemDB

This class is solely concerned with the SQLite database. It contains private properties *mydatareader*, *mycmd* and *myconn* for the internal implementation of functions. Also it contains helper functions which are used to access and query the database as SQLite API according to what is needed for the project. The methods *AddClient()*, *DeleteClient()*,

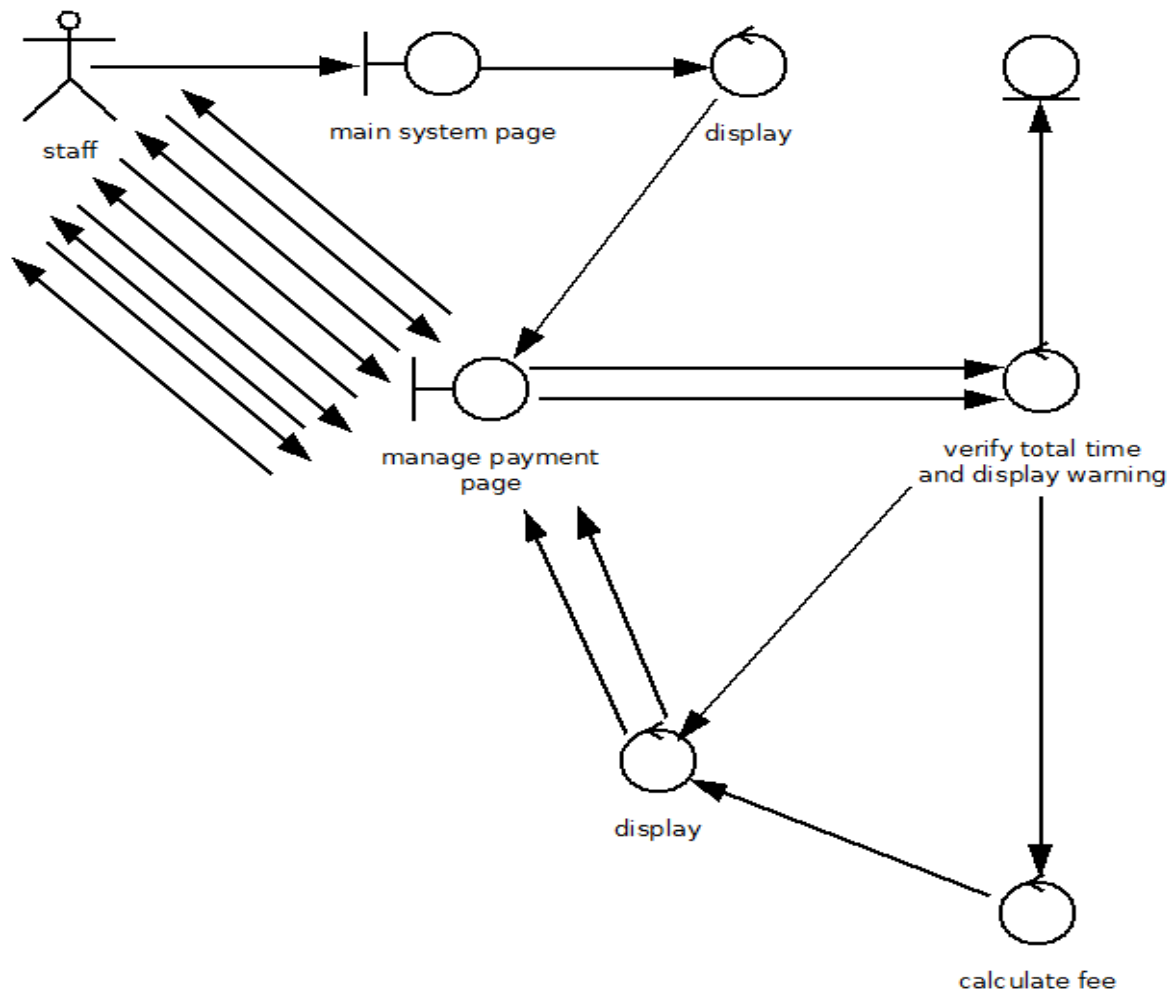
*UpdateClient()*, *SelectClient()* and *ViewAllClients()* were defined.

## **Robustness Analysis and Sequence Diagram**

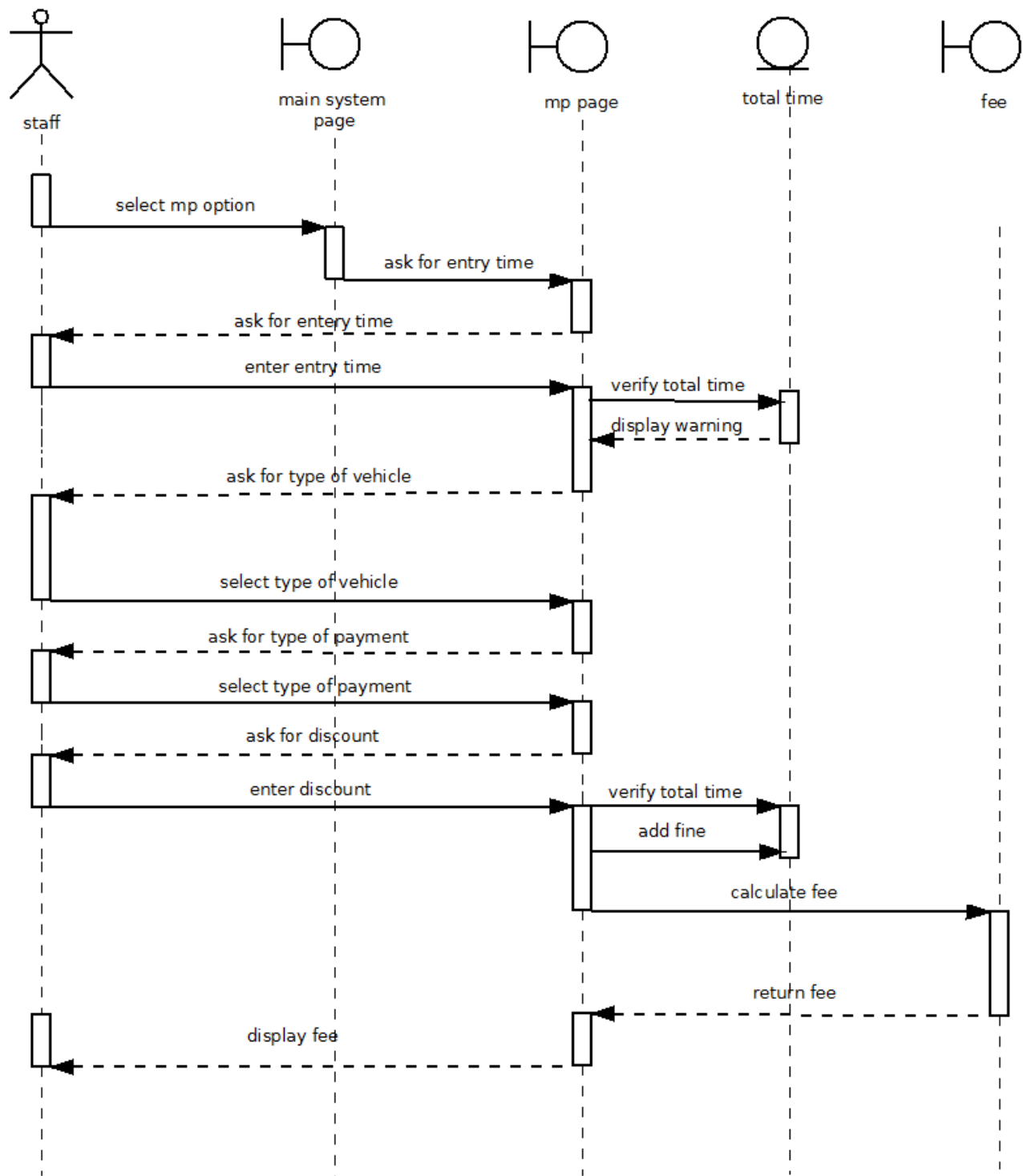
In the following, I performed the robustness analysis of the use case UC1 since it is the most interesting one from the stand point of interaction, while the other use cases are very simple and similar.



This is what resulted after the unification of the individual diagrams:



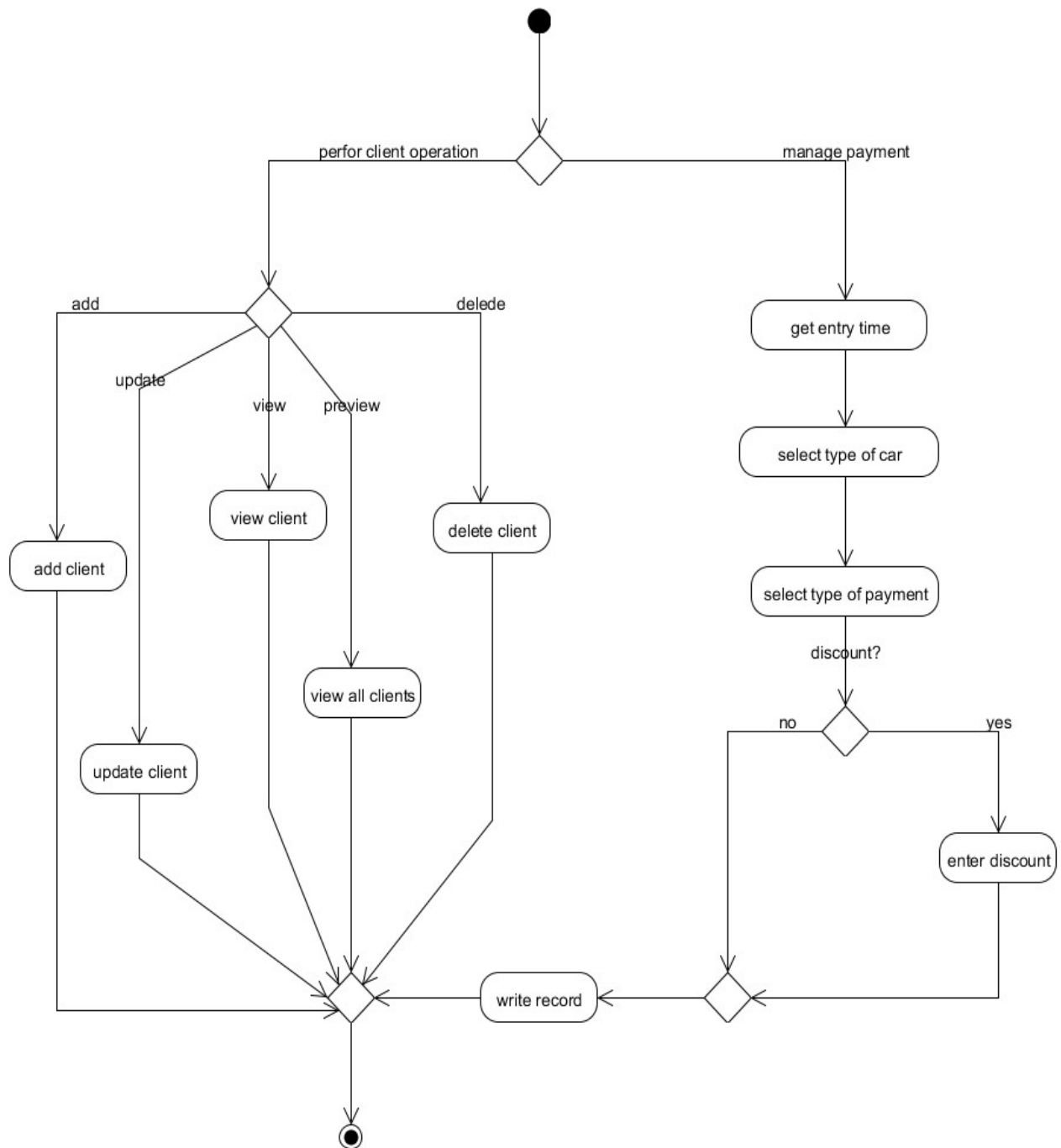
From the robustness diagram we can obtain the following sequence interaction diagram:



mp option: manage payment option

## State Diagrams

In order to better describe the behavior of the application in relation to the use cases outlined above, we use the state diagram. This diagram represents the states in which the application can be found in the course of its execution.



## IMPLEMENTATION

## File Main.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace ParkingLotSystem
{
    /*implimentation of program, the main class*/
    public class Program
    {
        public static void Main(string[] args)
        {
            /*variable declarations*/
            double rate;
            double discount;           // discount variable
            double totalTime;          //total time used
            double amount = 0.0;
            double amountDue = 0.0;    //represents fee calculated
            double amph = 0.0;         //constant amount/hour for each type of car
            double addamph = 0.0;      //constant additional amount/hour after the first hour
            string vehicle;             //string character to select the type of vehicle
            string answer;              //control variable

            Transaction transact = new Transaction();           //instantiated transaction
            WriteRecord wr = new WriteRecord();                 //instantiated writerecord
            ParkingLotSystemDB db = new ParkingLotSystemDB();   //instantiated
parkinglotsystemdb
            Console.WriteLine("\t\t===== PARKING LOT CASHIER =====");
            Console.WriteLine("\n\t\tSelect An Operation\n");
            Console.Write("\t\tM: Manage Payment C: Client Operation\t\t[ ]\b\b");
            string operation = Console.ReadLine(); //control variable to seclct an
operation at main prompt
            /*manage payment operation selection*/
            if (operation == "m")
            {
                /*do loop to control the fee operation*/
                do
                {
                    totalTime = transact.UsedTime(); //calling UsedTime function an
assigning its value to totalTime
                    discount = transact.Isdiscount(); //calling Isdiscount function,
assigning its value to discount
                    rate = transact.TypeOfPayment(); //calling TypeOfPayment, assinging it
to rate
                    amount = transact.CalculateFee(totalTime, discount, rate); //calling
CalculateFee, assigning its value to amount
                    Console.WriteLine("\n\tSelect type of vehicle.. C: Car V: Van B: Bus
T: Truck");
                    vehicle = Convert.ToString(Console.ReadLine());
                    /*switch control to switch between the type of car object*/
                    switch (vehicle)
                    {

```



```

        case "c":
            vehicle object c
            Vehicle c = new Vehicle("Car", 6.0, 3.0); //instantiating new

            Console.WriteLine("\tYou selected a " + c.Name);
            amph = c.AmountPerHour;
            addamph = c.AdditionalAmountPerHour;
            if (totalTime == 1)
            {
                amountDue = amount + amph;
            }
            else if (totalTime > 1)
            {
                addamph = (totalTime * addamph) - addamph;
                amountDue = amount + amph + addamph;
            }

            Console.WriteLine("\t\t=====");
            Console.WriteLine("\n\t\t\tTotal Fee: " + amountDue + "euros");

            Console.WriteLine("\n\t\t=====");
            break;
        case "v":
            vehicle object v
            Vehicle v = new Vehicle("Van", 6.5, 3.2); //instantiating new

            Console.WriteLine("\tYou selected a " + v.Name);
            amph = v.AmountPerHour;
            addamph = v.AdditionalAmountPerHour;
            if (totalTime == 1)
            {
                amountDue = amount + amph;
            }
            else if (totalTime > 1)
            {
                addamph = (totalTime * addamph) - addamph;
                amountDue = amount + amph + addamph;
            }

            Console.WriteLine("\t\t=====");
            Console.WriteLine("\n\t\t\tTotal Fee: " + amountDue + "euros");

            Console.WriteLine("\n\t\t=====");
            break;
        case "b":
            vehicle object b
            Vehicle b = new Vehicle("Bus", 7.0, 3.8); //instantiating new

            Console.WriteLine("\tYou selected a " + b.Name);
            amph = b.AmountPerHour;
            addamph = b.AdditionalAmountPerHour;
            if (totalTime == 1)
            {
                amountDue = amount + amph;
            }
            else if (totalTime > 1)
            {
                addamph = (totalTime * addamph) - addamph;
                amountDue = amount + amph + addamph;
            }

            Console.WriteLine("\t\t=====");
            Console.WriteLine("\n\t\t\tTotal Fee: " + amountDue + "euros");

            Console.WriteLine("\n\t\t=====");
            break;
        case "t":
            Vehicle t = new Vehicle("Truck", 10.0, 4.0); //instantiating

```

new vehicle object t

```
        Console.WriteLine("\tYou selected a " + t.Name);
        amph = t.AmountPerHour;
        addamph = t.AdditionalAmountPerHour;
        if (totalTime == 1)
        {
            amountDue = amount + amph;
        }
        else if (totalTime > 1)
        {
            addamph = (totalTime * addamph) - addamph;
            amountDue = amount + amph + addamph;
        }

        Console.WriteLine("\t\t=====");
        Console.WriteLine("\n\t\t\tTotal Fee: " + amountDue + "euros");

        Console.WriteLine("\n\t\t=====");
        break;
    }
    wr.PrintRecord(totalTime, amountDue); //calling PrintRecord method to
write the record to a file
    Console.WriteLine("\n\t\tDo You Want To Manage Another Payment?");
    Console.WriteLine("\n\t\tY: Yes N: No \t\t[ ]\b\b");
    answer = Console.ReadLine();
} while (answer == "y");
if (answer == "n")
    Console.WriteLine("\n\n\t\tOk Then, Bye!");
else{
    Console.WriteLine("\n\t\tYou Hit The Wrong Key.. Program Will End!");
}

/*Client operations selection*/
else if (operation == "c")
{
    Console.WriteLine("\n\t\tWhat Do You Want To Do?");
    Console.WriteLine("\n\t\tV: View Client\n\t\tA: Add Client\n\t\tD: Delete
Client\n\t\tU: Update Client\n\t\tP: Preview All Clients\t\t[ ]\b\b");
    string clientOperation = Console.ReadLine();
    /*switching between the type of operation*/
    switch (clientOperation)
    {
        case "v":
            db.SelectClient(); //view client information
            break;
        case "a":
            db.AddClient(); //add a new client to the system
            break;
        case "d":
            db.DeleteClient(); //delete existing client from the system
            break;
        case "u":
            db.UpdateClient(); //update an existing client in the system
            break;
        case "p":
            db.ViewAllClients(); //preview all the clients on the system
            break;
    }
}

Console.ReadKey();//read any key to end the program
}
```

## File CalculateFee.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace ParkingLotSystem{
    public class Transaction{
        /*variable declarations*/
        private int fine;           //integer variable for fine to be added to fee
        private DateTime exitTime; //current system time used for the exit time
        private TimeSpan entryTime; //time span variable used for exit time

        /*definition of public methods as used to access properties*/
        public TimeSpan EntryTime
        {
            get { return entryTime; }
            set { entryTime = value; }
        }
        public DateTime ExitTime
        {
            get { return exitTime; }
            set { exitTime = value; }
        }

        /*definition of the used time method*/
        public double UsedTime()
        {
            exitTime = DateTime.Now; //current system time
            Console.WriteLine("\n\tGet Entry Time In Format: '00:00:00'
\t\t[ ]\b\b\b\b\b\b\b\b\b\b");
            entryTime = TimeSpan.Parse(Console.ReadLine());
            double totalTime = (double)(exitTime - entryTime).Hour;
            Console.WriteLine("\n\tEntry Time: " + (entryTime) + "\n\tExit Time: " +
(exitTime) + "\n\tTotal Time: " + totalTime);
            if(totalTime > 12.0)
            {
                Console.WriteLine("\tWARNING!! Time Is More Than A Day.. You Will Pay A Fine");
            }
            return totalTime; //return value for total time used
        }

        /*definition of discount method*/
        public double Isdiscount()
        {
            double discount = 0.0;
            string isDiscount;
            Console.WriteLine("\n\tSpecial discount available?\n\t" + "Y: Yes, N: No
\t\t\t\t\t[ ] \b\b\b\b");
            isDiscount = Convert.ToString(Console.ReadLine()); //discount in percentage
            switch (isDiscount)
            {
                case "y":
                    Console.WriteLine("\tEnter Discount In % \t\t\t\t [ ]\b\b\b\b");
                    /* numerical validation */
                    while (!double.TryParse(Console.ReadLine(), out discount))
                    {

```

```

        Console.WriteLine("Invalid! Discount Must Be A Number");
        Console.Write("\tPlease Enter Discount In % \t\t\t [ ]\b\b\b\b");
    }
    Console.WriteLine("\tYou have given out a " + discount + "% discount");
    discount = discount / 100; //real value of discount
    break;
    case "n":
        Console.WriteLine("\tOk.. no special discount");
        break;
    }
    return discount;
}

/*definition for the type of car*/
public double TypeOfPayment()
{
    double rate = 0.0; //fixed amount of rate for each hour of time
    double sRate = 5.0; //standard rate per hour
    double eRate = 6.4; //event rate per hour
    string paymentType; //character string to select the type of payment

    Console.Write("\n\tSelect type of payment:\n\t S: Standard Rate\n\t E: Event
Rate\n\t\t\t\t\t [ ]\b\b");
    paymentType = Convert.ToString(Console.ReadLine());
    switch (paymentType)
    {
        case "s":
            Console.WriteLine("\tYou have selected a standard type of payment..");
            rate = sRate;
            break;
        case "e":
            Console.WriteLine("\tYou have selected an event type of payment..");
            rate = eRate;
            break;
    }
    return rate; //returned amount of rate
}

/*definition for the fee calculation method*/
public double CalculateFee(double totalTime, double discount, double rate)
{
    this.fine = 25;
    double totalAmount;
    totalAmount = totalTime * rate;
    discount = totalAmount * discount;
    totalAmount = totalAmount - discount;
    if (totalTime > 12)
    {
        totalAmount = totalAmount + fine;
    }
    return totalAmount; //return of total amount calculated
}
}
}

```

## **File Vehicle.cs**

```

using System;
using System.Collections.Generic;
using System.Linq;

```

```

using System.Text;
using System.Threading;
namespace ParkingLotSystem{
    /*the vehicle structure*/
    public struct Vehicle{
        private string name;           //name given to the vehicle type
        private double amountPerHour; //amount for first hour used for parking
        private double additionalAmountPerHour; //additional amount charged for each hour
after the first
        /*definition of public methods as used to access properties*/
        public string Name
        {
            get{ return name; }
            set{ name = value; }
        }
        public double AmountPerHour
        {
            get{ return amountPerHour; }
            set{ amountPerHour = value; }
        }
        public double AdditionalAmountPerHour
        {
            get { return additionalAmountPerHour; }
            set { additionalAmountPerHour = value; }
        }
        /* Constructor for vehicle structure*/
        public Vehicle(string _name, double _amountPerHour, double _additionalAmountPerHour)
        {
            name = _name;
            amountPerHour = _amountPerHour;
            additionalAmountPerHour = _additionalAmountPerHour;
        }
    }
}

```

### **File ParkingLotSystemDB.cs**

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Finisar.SQLite;
namespace ParkingLotSystem{
    public class ParkingLotSystemDB{
        /*variable declarations*/
        private SQLiteDataReader mydatareader = null;
        private SQLiteCommand mycmd = new SQLiteCommand();
        private SQLiteConnection myconn;
        public string cnn = "Data Source=C:/Users/iczc00l/Desktop/Seng
prj/Test/Test/bin/Debug/database.db;Version=3;New=False;Compress=False;"; //path to database
file

        /*definition of AddClient method*/
        public void AddClient()
        {
            myconn = new SQLiteConnection(cnn); //new connection object
            myconn.Open();
            mycmd = myconn.CreateCommand();
            Console.WriteLine("\n\n\t\tHow many clients do you want to register?\t[ ]\b\b");
            int numOfClients = 0;
            /* numerical validation */
            while (!int.TryParse(Console.ReadLine(), out numOfClients))

```

```

{
    Console.WriteLine("\t\tInvalid!! Must Be A Number");
    Console.Write("\n\n\t\tPlease How Many Clients Do You Want To Register?
[ ]\b\b");
}
/*for loop to control number of clients to be added*/
for (int i = 1; i <= numOfClients; i++)
{
    Console.Write("\n\n\t\tClient {0}'s ID is: \t\t\t\t\t[ ]\b\b\b\b", i);
    int id = 0;
    /* numerical validation */
    while (!int.TryParse(Console.ReadLine(), out id))
    {
        Console.WriteLine("\t\tInvalid! ID Must Be A Number");
        Console.Write("\n\n\t\tEnter Client {0}'s ID: \t\t\t\t\t[ ]\b\b\b\b",
i);
    }
    Console.Write("\n\n\t\tClient {0}'s Name is:
\t\t\t\t\t[ ]\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b", i);
    string name = Console.ReadLine();
    /*input validation*/
    while (string.IsNullOrEmpty(name))
    {
        Console.WriteLine("\t\tYou Hit A Wrong Key Or Must Not Include Number!");
        Console.Write("\n\n\t\tPlease Enter {0}'s Name : \t\t[
]\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b", i);
        name = Console.ReadLine();
    }
    /*getting the parking period*/
    Console.Write("\n\n\t\tClient {0}'s Parking Period
is:\t\t\t\t[ ]\b\b\b\b\b\b\b\b\b\b", i);
    string parkingperiod = Console.ReadLine();
    /*input validation*/
    while (string.IsNullOrEmpty(parkingperiod))
    {
        Console.WriteLine("\t\tParking Period Is Not Valid!");
        Console.Write("\n\n\t\tPlease Enter Client {0}'s Parking Period: [
]\b\b\b\b\b\b\b\b\b\b\b\b\b\b", i);
        parkingperiod = Console.ReadLine();
    }
    /*gettint the type of vehicle*/
    Console.Write("\n\n\t\tClient {0}'s Vehicle is:
\t\t\t\t\t[ ]\b\b\b\b\b", i);
    string vehicle = Console.ReadLine();
    while (string.IsNullOrEmpty(vehicle))
    {
        Console.WriteLine("\t\tInvalid Input!");
        Console.Write("\n\n\t\tPlease Enter Client {0}'s Vehicle:
\t\t\t\t\t[ ]\b\b\b\b\b", i);
        vehicle = Console.ReadLine();
    }

    /*gettint the entry time*/
    Console.Write("\n\n\t\tClient {0}'s Entry Time is:
\t\t\t\t\t[ ]\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b", i);
    string entrytime = Console.ReadLine();
    while (string.IsNullOrEmpty(entrytime))
    {
        Console.WriteLine("\t\tInvalid Input!");
        Console.Write("\n\n\t\tPlease Enter Client {0}'s Entry Time: \t\t[
]\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b", i);
        entrytime = Console.ReadLine();
    }
    /*command to insert into database*/
    var addClient = String.Format("INSERT INTO clients ([ID], [Name],

```

```

[Parking_Period], [Vehicle], [Entry_Time]]" +
                                "VALUES ('{0}', '{1}', '{2}', '{3}', '{4}')" , id,
name, parkingperiod, vehicle, entrytime);
        mycmd.CommandText = addClient;
        mycmd.ExecuteNonQuery();
    }
    myconn.Close(); //close the database connection
}

/*definition for SelectClient method*/
public void SelectClient()
{
    Console.WriteLine("\n\n\t\tPlease Enter An ID Number: [    ]\b\b\b\b");
    int id = 0;
    /*input validation*/
    while (!int.TryParse(Console.ReadLine(), out id))
    {
        Console.WriteLine("\t\tInvalid! ID Must Be A Number..");
        Console.WriteLine("\n\n\t\tPlease Enter An ID Number: [    ]\b\b\b\b");
    }
    /*command to perform database selection*/
    var selectClient = String.Format("SELECT * FROM clients" +
                                     " WHERE [ID] = '{0}'" , id);
    myconn = new SQLiteConnection(cnn);
    myconn.Open();
    mycmd = myconn.CreateCommand();
    mycmd.CommandText = selectClient;
    mycmd.ExecuteNonQuery();
    mydatareader = mycmd.ExecuteReader();
    /*while loop to read and display all the data on a row*/
    while (mydatareader.Read())
    {
        Console.WriteLine();
        Console.WriteLine("\n\n\n");

        Console.WriteLine("\t-----\n\n");
        Console.WriteLine("\t\t");
        Console.WriteLine(mydatareader["ID"]);
        Console.WriteLine("\t");
        Console.WriteLine(mydatareader["Name"]);
        Console.WriteLine("\t");
        Console.WriteLine(mydatareader["Parking_Period"]);
        Console.WriteLine("\t");
        Console.WriteLine(mydatareader["Vehicle"]);
        Console.WriteLine("\t");
        Console.WriteLine(mydatareader["Entry_Time"]);

        Console.WriteLine("\n\n\t-----");
    }
    myconn.Close(); //close the database connection
}

/*the View client method*/
public void ViewAllClients()
{
    var viewAllClients = "SELECT * FROM clients"; //select command to query database
    myconn = new SQLiteConnection(cnn); //create a new connection object
    try
    {
        myconn.Open(); //open connection
        mycmd = myconn.CreateCommand();
    }
}

```

```

mycmd.CommandText = viewAllClients;
mycmd.ExecuteNonQuery();
mydatareader = mycmd.ExecuteReader();
Console.WriteLine("\n\n");
Console.WriteLine(
    "ID          NAME          PARKING PERIOD          ENTRY TIME          VEHICLE");
Console.WriteLine(
    "-----");
/*while loop to read and display all the data on a row*/
while (mydatareader.Read())
{
    /*getting the results of each column*/
    Int64 _id = (Int64)mydatareader["ID"];
    string _name = (string)mydatareader["Name"];
    string _parkingperiod = (string)mydatareader["Parking_Period"];
    string _vehicle = (string)mydatareader["Vehicle"];
    string _entrytime = (string)mydatareader["Entry_Time"];

    /* printing out the results */
    Console.Write("{0,-10}", mydatareader["ID"]);
    Console.Write("{0,-20}", _name);
    Console.Write("{0,-40}", _parkingperiod);
    Console.Write("{0,-60}", _vehicle);
    Console.Write("{0,-100}", _entrytime);
    Console.WriteLine();
}
}
finally
{
    //close the reader
    if (mydatareader != null)
    {
        mydatareader.Close();
    }

    //close the connection
    if (myconn != null)
    {
        myconn.Close();
    }
}

/*definition of Udate Client method*/
public void UpdateClient()
{
    myconn = new SQLiteConnection(cnn);
    myconn.Open();

    Console.Write("\n\t\t\tWhat Are You Updating? \n\t\t\t N:Name \n\t\t\t P: Parking
Period \n\t\t\t V: Vehicle \n\t\t\t T: Entry Time \n\t\t\t [ ]\b\b");
    string updatetype = Console.ReadLine();
    /*input validation to update either name, parking period, vehicle or entry
time*/
    while (((updatetype != "n") && (updatetype != "p") && (updatetype != "v") &&
(updatetype != "t")) && ((updatetype != "N") && (updatetype != "P") && (updatetype != "V")
&& (updatetype != "T")))
    {
        Console.WriteLine("\n\t\t\tInvalid Selection For Update!!");
        Console.Write("\n\t\t\tWhat Are You Updating? \n\t\t\t N:Name \n\t\t\t P: Parking
Period \n\t\t\t V: Vehicle \n\t\t\t T: Entry Time \n\t\t\t [ ]\b\b");
        updatetype = Console.ReadLine();
    }
}

```



[illegible]

[illegible]

[illegible]



```

prj/writeRecord.txt", true))
    {
        writer.Write(DateTime.Now + "\n");
        writer.Write("Used Time: \t" + _usedTime + "\n");
        writer.Write("Amount due: \t" + _amountDue + "\n\n");
    }
}
}
}
}

```

## TESTING

To test the program I preferred the use of the validation tests, ie tests used especially in the case in which the program is at its first release. These tests are not completely exhaustive as it is in the real world, but is deeply made by White Box testing methods, ie those methods where it is examined the internal structure of the system, and of methods of Black Box Testing conversely that do not take into account the internal structure the program but take into consideration input validations, thereby determining the correct output of the program.

The following tests were performed with special care about the Use Cases in the specification, with the control of all the cases relating to, potential errors, and their rise to its treatment of exceptions. The following are some tests, inherent in the various use cases of use of the program:

*Test1 on UC1:*

```
file:///C:/Users/iczcool/Desktop/Seng prj/Test/Test/bin/Debug/Test.EXE

===== PARKING LOT CASHIER =====

Select An Operation

M: Manage Payment C: Client Operation          [m]

Get Entry Time In Format: '00:00:00'           [06:00:00]

Entry Time: 06:00:00
Exit Time: 27-Jun-14 13:52:30
Total Time: ?

Special discount available?
Y: Yes, N: No                                  [y]
Enter Discount In %                             [10 ]
You have given out a 10% discount

Select type of payment:
S: Standard Rate
E: Event Rate                                  [e]
You have selected an event type of payment..

Select type of vehicle.. C: Car U: Van B: Bus T: Truck
You selected a Bus
=====

Total Fee: 70.12euros

=====

Do You Want To Manage Another Payment?
Y: Yes N: No                                  [4]

You Hit The Wrong Key.. Program Will End!
```

Test2 on UC1:

```
file:///C:/Users/iczcool/Desktop/Seng prj/Test/Test/bin/Debug/Test.EXE

===== PARKING LOT CASHIER =====

Select An Operation

M: Manage Payment C: Client Operation          [m]

Get Entry Time In Format: '00:00:00'           [12:30:00]

Entry Time: 12:30:00
Exit Time: 27-Jun-14 13:37:47
Total Time: 1

Special discount available?
Y: Yes, N: No                                  [n]
Ok.. no special discount

Select type of payment:
S: Standard Rate
E: Event Rate                                  [s]

You have selected a standard type of payment..

Select type of vehicle.. C: Car U: Van B: Bus T: Truck

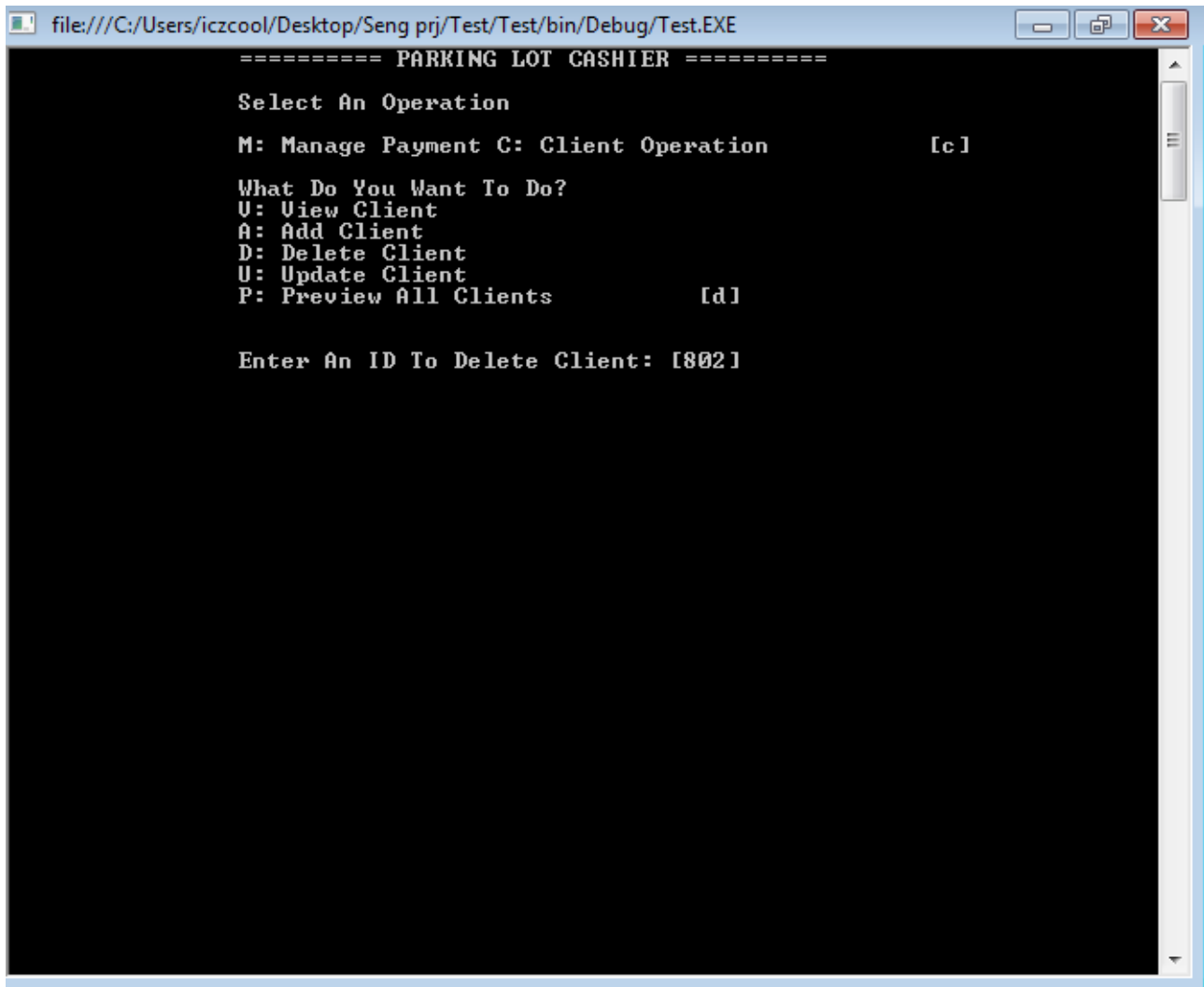
You selected a Van
=====

Total Fee: 11.5

=====

Do You Want To Manage Another Payment?
Y: Yes N: No                                  [_]
```

Test1 on UC3:



```
file:///C:/Users/iczcool/Desktop/Seng prj/Test/Test/bin/Debug/Test.EXE

===== PARKING LOT CASHIER =====

Select An Operation

M: Manage Payment C: Client Operation          [c]

What Do You Want To Do?
U: Uiew Client
A: Add Client
D: Delete Client
U: Update Client
P: Preview All Clients                        [d]

Enter An ID To Delete Client: [802]
```

*After the operation, UC6 shows;*



```
file:///C:/Users/iczcool/Desktop/Seng prj/Test/Test/bin/Debug/Test.EXE
file:///C:/Users/iczcool/Desktop/Seng prj/Test/Test/bin/Debug/Test.EXE

===== PARKING LOT CASHIER =====

Select An Operation

M: Manage Payment C: Client Operation          [c]

What Do You Want To Do?
U: View Client
A: Add Client
D: Delete Client
U: Update Client
P: Preview All Clients          [u]

What Are You Updating?
N: Name
P: Parking Period
V: Vehicle
T: Entry Time
[ ]

Enter Existing ID: [800]

Enter Existing Parking Period: [1week          ]

Enter Existing Vehicle: [car ]

Enter Existing Entry Time: [11:11:11      ]

Enter new Name: [Mike Tutu_          ]
```

```
file:///C:/Users/iczcool/Desktop/Seng prj/Test/Test/bin/Debug/Test.EXE

U: Update Client
P: Preview All Clients          [p]

ID      NAME      PARKING PPERIOD      ENTRY TIME      VEHICLE
-----
800     Mike Tutu     1week               11:11:11        car
801     Kelvin Aromana  2 months            12:36:00        car
802     Osei Linda     3 weeks             04:40:23        van
803     Mark Charway   1 month              13:16:27        truck
804     Prince Liam    2 weeks              15:06:40        van
805     Evans Boat     3 months             22:11:45        truck
```

An example history of write record file;

26-Jun-14 17:31:06

Used Time: 19

Amount due: 180

26-Jun-14 17:33:54

Used Time: 6

Amount due: 51

26-Jun-14 17:36:52

Used Time: 6

Amount due: 51

## **COMPILATION AND EXECUTION**

Instructions to run the program:

- Install Visual Studio Express 2012 (or 2010) for Windows Desktop.
- Open the file with a double click ParkingLotSysApplication.exe