AARHUS UNIVERSITY

AUTOMATED PARKING SYSTEM

Mini-project

TIMoMi

Modeling of Mission Critical Systems

Deadline: 23/05-2017

Morten Morberg Madsen

AU501465

Table of Contents

Document References

Reference	Document Description	Document Identifier
[DOC 1]	Appendix including project source code, tests and coverage.	AutomatedParkingSystem
[LINK 1]	Link for the Automated Parking System build underneath the cultural center of Aarhus called “Dokk1”	https://dokk1.dk/besog-dokk1/parkering
[LINK 2]	Wikipedia description of Danish license plates including rule sets of allowed combinations	https://da.wikipedia.org/wiki/Nummerplade

1 Introduction

This report describes the mini-project Automated Parking System developed in relation to the course Modeling of Mission Critical Systems. The mini-project is individually carried out, as a mandatory part of the course.

The purpose of the mini-project is to illustrate the use of the Vienna Development Method VDM++ which allows for modeling of object-oriented systems. To create a context for such a system, it was chosen to work on the Automated Parking System, which was considered valid as a mission critical system and contained aspects which could prove the strength of the modelling language VDM++. Since the scope of the project is to use major concepts of the language some examples from the project can be considered proof of concepts and could be executed differently.

The report contains an introduction to the project, containing the requirements which are should be met by for the system. Furthermore a discussion regarding the decisions and approaches chosen by the student will be presented. Hereafter a description of the key aspects of the developed model, concluded with a description of how the model could be expanded using VDM and finally a conclusion of the project. In addition the appendix of the report will include the source code, tests and results of the model.

1.1 System Description

Many cities are affected by the rising geographical centralization we feel in today's society. A result of this trend is the cramped up city centrums, which needs to transport and guest more people than ever before. One of the major issues involved in an increase of traffic in a specific area, is the need of parking spaces. To optimize the area needed for these parking spaces, underground Automated Parking Systems has become more and more used. For such a system to be worth building, it is clear that its reliability and dependability is of most importance. The consequences in failures is mission critical in that it either could involve loss of users properties in damaging cars, or in worst cases harm or cost human lives in the process of parking a car. When building such systems it is therefore necessary to ensure that the system functions as expected, and thus a model of the system would be well fitting. An example of such Automated Parking System is found under the cultural center of Aarhus named Dokk1 [LINK 1], which is the inspiration for this mini-project.

The system modeled in the mini-project is represented on the context diagram of [Figure 1], where the system is decomposed into its subcomponents.

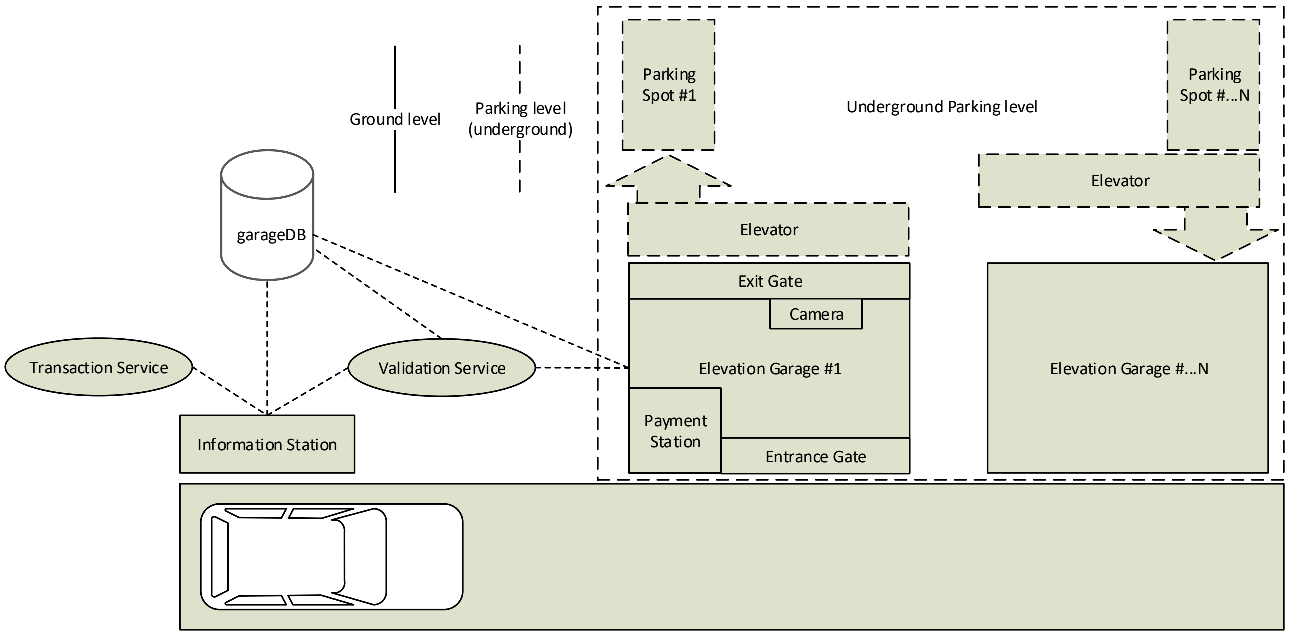


Figure 1 - Automated Parking System context diagram

The diagram represents both the ground level parts of the system, which is marked with the solid lines and the underground parts which is illustrated using the dashed lines.

To provide an understanding of how the system works, a short description is given:

When a user arrives at the Information Station, it will guide the user to a free Elevation Garage. After parking the car into the garage, the user must simply insert valid payment information to the Payment Station of an Elevation Garage. Here after the system handles the parking of the car in the underground Parking Level, and the user is free to continue his/her business. When the user arrives at the system to retrieve a parked car, the user inserts the payment information at the Information Station. The station will then withdraw the parking fee and guide the user to the Elevation Garage of which the car will be raised to. When the car is raised, the gates for the Elevation Garage opens so that the user can access the car and drive away though the exit gate.

2 Requirements

The initial steps in developing a formal model contains the defining, reading and analysing of the system requirements. This section will present the requirements and enlist them to the corresponding subcomponents of the Automated Parking System presented on [Figure 1]. This will hereby illustrate the responsibilities of the system components. In addition, general requirements for the entire system is also presented.

General

[REQ 01] A user shall be able to park a car in the system with either a valid credit card or a valid parking card.

[REQ 02] If a user wishes to park a car with a parking card, the associated car must have a parking permit.

[REQ 03] The parking rates of the system must change depending of the payment method and the nationality of the car, ranging from “Danish”, “European” or “Other”.

Information Station

[REQ 04] When a user arrives to the Information Station, the station shall guide the user to a free available Elevation Garage.

[REQ 05] When a user wishes to retrieve a car from the parking system, the Information Station shall be able to find the car associated to the payment information inserted by the user.

[REQ 06] When the correct car has been identified as described in **[REQ 05]** the Information Station shall guide the user to the Elevation Garage, where the car is to be retrieved.

[REQ 07] If more than one car has been registered in the system with the same payment information, the user shall be able to choose which car to retrieve by choosing the license plate information of the desired car.

[REQ 08] When a retrieval of a car is initiated, the parking fee must be calculated and withdrawn from the payment information. The fee must be based on the parking duration and the nationality of the car.

Elevation Garage

[REQ 09] An Elevation Garage shall be able to validate the license plate of a parked car, by using a camera and a validation service.

[REQ 10] An Elevation Garage shall be able to validate the payment information inserted by a user, by using a payment station and a validation service.

[REQ 11] The entrance and exit gates of an Elevation Garage shall be handled such that:

1. A free garage can only be accessed through the entrance gate.
2. The entrance and exit gates must be closed when a car is lowered to or elevated from the parking level.
3. When a car is to leave an Elevation Garage, both gates shall open so that the user can access the car and leave the garage.
4. If validation is failed, the gates must open as described in “**3.**”.

[REQ 12] When a car and the associated payment information has been validated, the Elevation Garage shall be able to register the payment information, license plate and parking time of the car into a garage database.

Parking Level

[REQ 13] The Parking Level shall be able to park a car on a free parking spot, and retrieve a car based on the license plate.

3 Abstractions and approach

The following section describes the decisions made by the student both in regards of abstractions, and the approach of the modeling.

3.1 Abstractions

To define a base system of which the model will be created upon, it was needed to abstract aspects of a real-life system away from the modelled system. The abstraction of the model is done either because of unnecessary complexities or to avoid "more of the same" scenarios which is considered needless for the mini-project.

The following elements is decided to be abstracted away from the modelled system:

Table 1 – Main Abstractions

Element:	**Reason and chosen solution:**
Size restrictions of cars	The physical restrictions of which vehicle that can be parked has not been considered in the scope of the project. Real autonomous parking systems both measure and weight cars before accepting them, to ensure that the underground system can handle the car. Instead the validation process of the built model only accepts vehicles with license plates in the domain range of cars.
Complexity of credit cards	The representation of credit cards in the system is simplified to be a number series of the length 6, and the storing of the payment information is not applied with any sort of encryption or other protection aspects.
Amount of parking spots	It has been decided to not include a limit of possible parking spots in the underground parking level. The addition of this would not add any interesting aspects other than that users would either leave the system or wait in a queue for a free spot.
Intelligent garage and parking level	Some of the logic needed for the system to function, has been placed in "non intelligent" components. An elevation garage is considered to use a camera and payment station to enable validations, but can do this by the use of a validation service. In addition the parking level is responsible of act both as the elevator responsible of levelling cars as well as a sled able to move cars to parking spots.
Sequential solution	As concurrency has not been added to the final result, the concurrent processes of an Automated Parking System has not been modelled. Instead actions has been made to allow for this addition, which is described further in section Future work].

Of course other abstractions is added to the final solution, but these are considered the most noteworthy.

3.2 Modelling approach

Many approaches can be taken when building a model, depending on the system different approaches can be deemed most fitting.

For a system as the Automated Parking System, where user's interaction with the system is of most importance, the initial approach has been to analyze the system flow from a user's perspective. This resulted in a list of possible classes and operations needed for the system to meet the user's goal. When the list of classes and operations were created, a specification of the requirements for the individual subsystems was defined. Next the associations and dependencies of the different classes were determined to create the structure of the system. Then the individual operations and functions were implemented as needed, and lastly tested using VDMUnit to ensure their individual requirements was meet. When all system classes were individually tested, the next step included a complete integration test for the whole system using a scenario meeting the systems different critical functionalities.

Also the abstraction level for the approach should be settled on. Here it was determined to model the system as a sequential model, including the traditional classes for such a model using VDM, such as World, Timer and Enviroment, all used to create a base for creating stimuli for the system, and let the system respond as expected.

4 Automated Parking System

The complete Automated Parking System is illustrated on the class diagram represented on [Figure 2]. The class diagram was created using Modelio by importing an UML transformation of the implemented VDM++ project into Modelio. Hereafter the different classes were added to the diagram and only key elements such as noteworthy associations and member variables is represented.

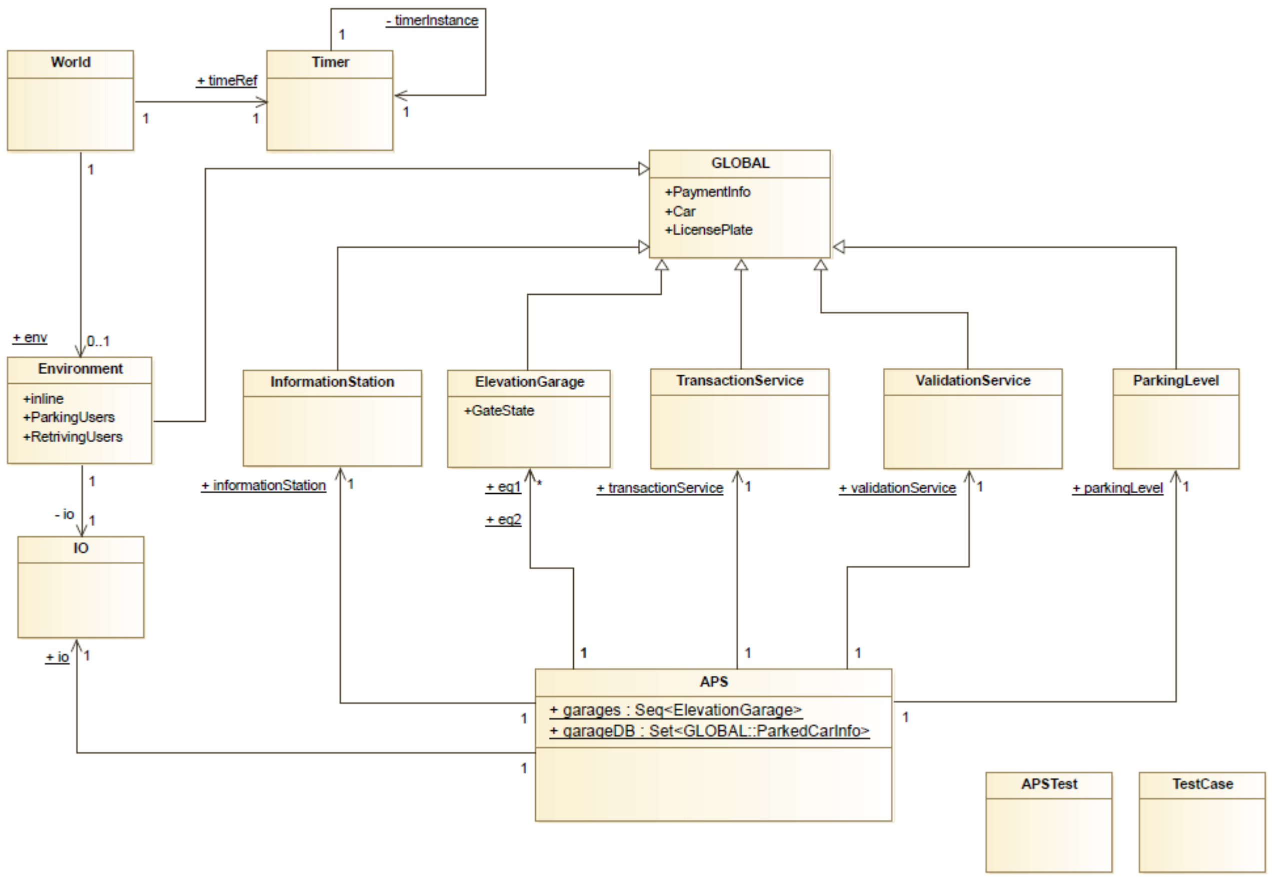


Figure 2 - Automated Parking System class diagram

The structure of the final system is based upon the typical design structure of sequential design models in VDM. In the top left corner of the class diagram, the World class is located, which is responsible of setting up the environment and initialize the system as well as execution of them. The World uses a Timer class to access a notion of time, used for the time-critical parts of the system such as the calculation of parking fee. The Environment class is responsible of creating a signal to the system which defines the stimuli that the system reacts to. The flow of the system is thus handled by the Environment. The remaining classes defines the Automated Parking System itself. The APS (Automated Parking System) is the overall system class, normally defined as a *SystemName* class. It is responsible of creating static public instances for all system components and thereby defines the initialized system. The following section will describe the remaining system component classes as well as the GLOBAL class.

4.1 GLOBAL

The GLOBAL class is a *superclass*, responsible of providing the general types and definitions used around the system as well as in the Environment class.

By introducing the GLOBAL class, the domain of different system-specific definitions is defined in a structured way. Instead of defining these types, which are broadly used all over the system, within each component, the individual components instead inherits from the GLOBAL superclass using the following syntax:

```
class ElevationGarage is subclass of GLOBAL
```

Listing 1 - Subclassing

The first type which was needed to be defined within the GLOBAL class, was the types needed to be able to model the car and its license plate:

```
Types

public CountryIdentifier = <Danish> | <European> | <Other>;

public RegistrationLetters = String
inv rl == len rl = 2;

public NumberSeries = nat
inv num == num <= 99999 and num >= 5000;

public LicensePlate :: ci : CountryIdentifier
                       rl : RegistrationLetters
                       ns : NumberSeries;

public Car :: licensePlate : LicensePlate
              parkingPermit : bool;
```

Listing 2 – Car types

The first type is the CountryIdentifier, which is a *Quote Type* which can be seen as an enumeration type used in other programming languages. The individual quotes is a part of a union type, meaning that the CountryIdentifier either are Danish, European or Other. These are later in the system used to determine the nationality of the car, to decide upon the parking fee.

The next type is a RegistrationLetters, which is of the type *String*. In VDM++ String is not a default type, and therefore to be able to use String, it must be defined as a *sequence of char* as follows:

```
public String = seq of char;
```

Listing 3 - String

The RegistrationLetters consists an *Invariant* defining that the length of the RegistrationLetters only can be 2 chars.

Next the NumberSeries is defined using a *nat*, these are limited to be numbers from 5000 to 99999 to make sure they are in the range of regular license plates for cars matching the description in [LINK 2].

Finally the LicensePlate can now be defined as a *Record Type* which is a *Compound Type* that can consists of multiple types. Meaning that a LicensePlate is now build upon a ContryIdentifier, a Registrationletters and a NumberSeries.

Lastly another Composite Type is defined, namely the Car which also holds a parkingPermit, of the type *bool*. This is used to identify weather a car can be parked using a ParkingCard or not.

The remaining types within the GLOBAL class is represented in the following table:

Name	Type	Description
`CreditCard`	`seq of nat`	Defining one of the possible payment methods allowed in the system. Combined of a sequence of the length 6, each element is maximum the value of 9. To reduce the complexity regarding being able to read real credit cards, this sequence is used.
`ParkingCard`	`String`	The other payment possibility is the ParkingCard which is a string, or sequence of chars. The ParkingCard requires a car to contain a parkingPermit as described earlier.
`PaymentInfo`	`CreditCard | ParkingCard`	The two types of payment information allowed in the system.
`ParkingFee`	`nat`	The calculated parking fee to be withdrawn from the users CreditCard
`ParkedCarInfo`	`ParkedCarInfo :: License : LicensePlate Info : PaymentInfo parkTime : Time;`	Used as elements in the garage database to define which LicensePlate is parked with which PaymentInfo and at what Time.

4.2 APS

The APS class is as mentioned the overall system class, and contains the static public instances of the Automated Parking System:

```
class APS is subclass of GLOBAL

instance variables
               public static eg1 : ElevationGarage := new ElevationGarage();
               public static eg2 : ElevationGarage := new ElevationGarage();

               public static garages : seq of ElevationGarage := [eg1, eg2];

               public static informationStation : InformationStation := new InformationStation();

               public static parkingLevel : ParkingLevel := new ParkingLevel();

               public static validationService : ValidationService := new ValidationService();

               public static transactionService : TransactionService := new TransactionService();

               public static garageDB : set of ParkedCarInfo := {};

               public static io : IO := new IO();
end APS
```

Listing 4 – APS overall system class

What is noteworthy of this listing, is that the APS holds a sequence of ElevationGarages of which is later used to find a free garage. A sequence was founded as a fair use to model the row of valid garages available in such a system, where a set would not provide the same aspect of having a row of garages.

Also the APS class holds the garageDB which is defined as a set of the Composite Type ParkedCarInfo. This is the database illustrated on the context diagram [Figure 1] which is used for the system to know which car is parked. Since every car is checked to include a unique license plate, a set was considered valid as a database system, and because the order of the elements does not matter in the system context.

Lastly the APS holds an instance of the IO class, which is used by the InformationStation and ElevationGarage to simulate printing of receipts for the user, which is considered as the feedback of the system.

4.3 ElevationGarage

The ElevationGarage is considered as a crucial part of the system, especially in regards of the safety of the user.

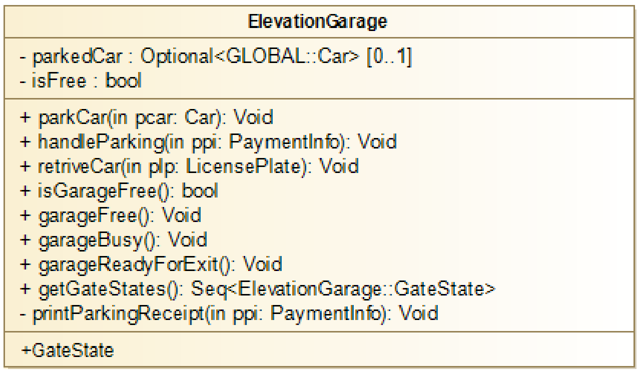

ElevationGarage
- parkedCar : Optional<GLOBAL::Car> [0..1]
- isFree : bool
+ parkCar(in pcar: Car): Void
+ handleParking(in ppi: PaymentInfo): Void
+ retriveCar(in plp: LicensePlate): Void
+ isGarageFree(): bool
+ garageFree(): Void
+ garageBusy(): Void
+ garageReadyForExit(): Void
+ getGateStates(): Seq<ElevationGarage::GateState>
- printParkingReceipt(in ppi: PaymentInfo): Void
+GateState

Figure 3 – ElevationGarage Class

To ensure that the user is not stuck within the parked car, and therefore would be elevated with the car, it is critical to ensure the states of the ExitGate and EntranceGate of each ElevationGarage is handled correctly. In the listing below, is shown how the states of the gates is handled by the *Quote Type* GateState. As well a garage consists of an *Optional Type* [Car] to indicate that it can hold a Car, and a *Boolean Type* to validate if the garage is free.

```
types

public GateState = <Open> | <Closed>;

instance variables
               EntranceGate : GateState := <Open>;
               ExitGate : GateState := <Closed>;
               parkedCar : [Car] := nil;
               isFree : bool := true;
```

Listing 5 – ElevationGarage types and instance variables

Because an ElevationGarage is considered free to use when instantiated, the default values of the gates is that the EntranceGate is open, and ExitGate is closed. This is done to allow a car to be parked within the garage, and still ensure a car is not parked in the garage from the wrong side.

```
public parkCar : Car ==> ()
parkCar(pcar) == (
               parkedCar := pcar;
               isFree := false;)
pre parkedCar = nil;
```

Listing 6 – parkCar operation

The parkCar operation is initiated by the user when a free ElevationGarage is found. The progress of getting an ElevationGarage assigned by the system will be described in section [4.6 InformationStation]. After the user has parked the car, the handleParking operation is called. This is seen as an abstraction of that the user has left the car and inserted a payment card into the payment station of the garage.

```
public handleParking:  PaymentInfo ==> ()
handleParking(ppi) == (
   if not isFree
     then (
        if APS`validationService.validateLicensePlate(parkedCar.licensePlate) and
           APS`validationService.validatePaymentInfo(ppi,parkedCar.parkingPermit)
```

```
            then (garageBusy();
                  APS`informationStation.registerCar(parkedCar.licensePlate,ppi,World`timeRef.getTime());
                  APS`parkingLevel.parkCar(parkedCar);
                  printParkingReceipt(ppi);
                  APS`io.print("STATUS: garageDB contains: ");APS`io.print(APS`garageDB);APS`io.print("\n");
                  parkedCar := nil;
                  garageFree();)
         else
         (garageReadyForExit();
          APS`io.print("STATUS: Error - <ValidationFailed>\n");
          parkedCar := nil;
          garageFree();)
          )
)
pre EntranceGate = <Open> and ExitGate = <Closed> and isFree = false;
```

Listing 7 – handleParking operation

First the operations checks *if* a car is parked, and then validates the LicesenPlate of the car, which is done in reality using a camera, and in the model by calling a ValidationService. Also the inserted paymentInfo is validated using the same service. By using an *if-then-else* statement, if both are validated with success, the parking of the car is started by first closing the garage gates by calling garageBusy, to ensure no user enters the garage. Hereafter the elevation of the car is done by the parkCar call from the parkingLevel. Next a receipt is printed for the user, which is done by calling a selection of io.prints. As the car has now been parked in the underground parkingLevel, the parkedCar of the ElevationGarage is removed by setting it as nil, and the gates is opened by the following operation:

```
public garageFree: () ==> ()
garageFree() == (
   if EntranceGate = <Closed>
      then EntranceGate := <Open>;
   if ExitGate = <Open>
      then ExitGate := <Closed>;
   isFree := true;)
pre EntranceGate = ExitGate and isFree = false;
```

Listing 8 – garageFree operation

garageFree is one of three operations ensuring the gate states of the garages is handled fitting the requirements. The reason for choosing the “if then” checks on the current gate states is to model the motors of the gates which would not be activated if in the right state already.

The *PreCondition* of the operation ensures that the garage can only be called from scenarios where the garage could become free, which is if the gates has the same state, and the garage is free.

4.4 ValidationService

The ValidationService is used by the ElevationGarage and the InformationStation to validate the user inputs. Two types of inputs can be validated by the service, a LicensePlate or a PaymentInfo, the following examples is used to handle the validation of the LicensePlate only.

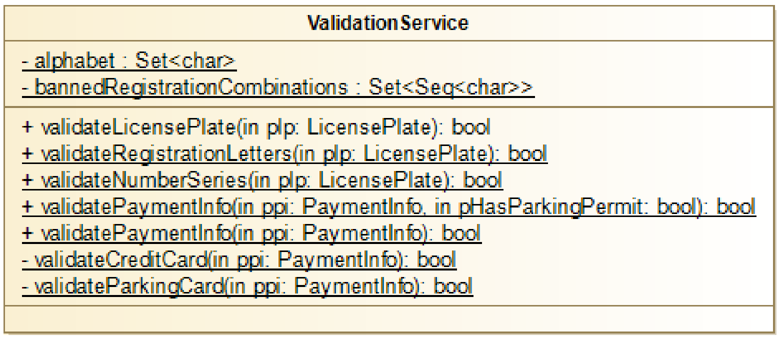

ValidationService
- alphabet : Set<char>
- bannedRegistrationCombinations : Set<Seq<char>>
+ validateLicensePlate(in plp: LicensePlate): bool
+ validateRegistrationLetters(in plp: LicensePlate): bool
+ validateNumberSeries(in plp: LicensePlate): bool
+ validatePaymentInfo(in ppi: PaymentInfo, in pHasParkingPermit: bool): bool
+ validatePaymentInfo(in ppi: PaymentInfo): bool
- validateCreditCard(in ppi: PaymentInfo): bool
- validateParkingCard(in ppi: PaymentInfo): bool

Figure 4 - ValidationService Class

According to Wikipedia [LINK 2] a set of restrictions is to be considered when validating license plates. The first restriction is set at the available letters used for the registrationLetters. To set up these restrictions, two values are defined in the ValidationService:

```
Values

alphabet = {'A','B','C','D','E','F','G','H','J','K','L','M','N','O','P','R','S','T','U','V','X','Y','Z'};

bannedRegistrationCombinations = {"BH", "BU", "CC", "CD", "DK", "DU", "EU", "KZ", "MU", "PY", "SS", "UD",
"UN", "VC"};
```

Listing 9 – ValidationService values

The first type is the valid alphabet containing the valid chars. The next set is banned combinations of the alphabet.

```
public validateRegistrationLetters : LicensePlate -> bool
validateRegistrationLetters(plp) ==
  plp.rl not in set bannedRegistrationCombinations and
  plp.rl(2) <> 'O'
pre plp.rl(1) in set alphabet and plp.rl(2) in set alphabet;
```

Listing 10 – validateRegistrationLetters function

The validateRegistrationLetters contains a *PreCondition* which stating that the two chars used on the licenseplate should be within the alphabet. Next the registrationsLetters of the LicensePlate is checked for not being in the set of bannedRegistrationCombinations. As well a check ensuring that the second element of the registrationLetters is not a ‘O’, is performed.

```
public validateLicensePlate : LicensePlate ==> bool
validateLicensePlate(plp) == (
   return validateRegistrationLetters(plp) and validateNumberSeries(plp) and
                   plp not in set {x.license | x in set APS`garageDB};);
```

Listing 11 – validateLicensePlate operation

The complete validation of the LicensePlate is done by the validateLicensePlate operation on as shown on [Listing 11 – validateLicensePlate operation]. Which also calls the validateNumberSeries ensuring that the numbers used on the LicensePlate is within the allowed range for the vehicle type of cars as presented in [Table 2 - Allowed LicensePlate types]

Table 2 - Allowed LicensePlate types

NumberSeries	**LicensePlate type**
5000-5399	Fixed test plates – cars
20000-75999	Passenger cars
76000-77999	Diplomat cars
98000-99999	Taxis, Rental Limousines etc.

As the physical restrictions of cars has not been considered as a scope of the project, this is the only form of validation on the type of vehicle in the system, where the case of Rental Limousines should be the only problematic type.

Lastly the validation uses a *Set Comprehension* to ensure that an element in the database does not contain the LicensePlate which is being validated, and thereby ensure unique license plates in the system.

4.5 ParkingLevel

The ParkingLevel is considered as an intelligent parking area which can handle parking of cars, and retrieval of a car from the ParkingLevel. The ParkingLevel could be modeled by multiple collection types such as *sets*, *sequences* and *maps*. To show the use of maps, it has been chosen as the collection type used.

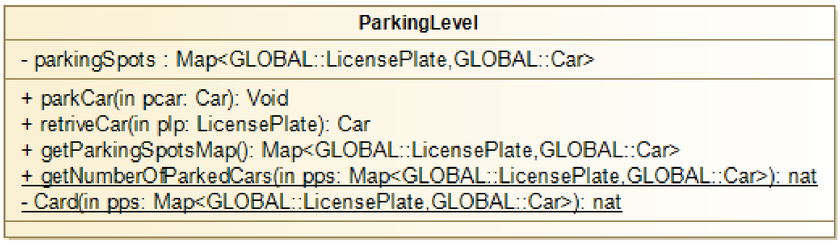

ParkingLevel
- parkingSpots : Map<GLOBAL::LicensePlate,GLOBAL::Car>
+ parkCar(in pcar: Car): Void
+ retriveCar(in plp: LicensePlate): Car
+ getParkingSpotsMap(): Map<GLOBAL::LicensePlate,GLOBAL::Car>
+ getNumberOfParkedCars(in pps: Map<GLOBAL::LicensePlate,GLOBAL::Car>): nat
- Card(in pps: Map<GLOBAL::LicensePlate,GLOBAL::Car>): nat

Figure 5 - ParkingLevel Class

The following listing shows the instance variable of the ParkingLevel, which holds the parkingSpots, defined as a mapping of LicensePlate to Car, and in that way the unique license plates is used to identify cars within the parkingLevel. The *Invariant* of the parkingSpots states that no car in the parkingLevel may hold the same licensePlate.

```
instance variables
parkingSpots : map LicensePlate to Car := {|->};
inv forall x,y in set rng parkingSpots & x <> y => x.licensePlate <> y.licensePlate
```

Listing 12 – ParkingLevel instance variables

The parkCar operation simply adds a mapping of a licensePlate to a Car to the parkingSpots using the *munion* operation. Again the *PreCondition* ensures for unique licensesPlates as the invariant mentioned above.

```
public parkCar : Car ==> ()
parkCar(pcar) ==
               parkingSpots := parkingSpots munion {pcar.licensePlate |-> pcar}
pre forall i in set rng parkingSpots & pcar.licensePlate <> i.licensePlate;
```

Listing 13 – parkCar operation

Of course it shall be possible to retrieve a car from the system, which is done using the following operation:

```
public retriveCar : LicensePlate ==> Car
retriveCar(plp) == (
               dcl car : Car := parkingSpots(plp);
               parkingSpots := {plp} <-: parkingSpots;
               return car;)
pre plp in set dom parkingSpots;
```

Listing 14 – retriveCar operation

The operation uses a *declaration* to get a hold of the Car which is wished received, before it is removed using the Domain Restriction operator <-:. The declared Car is returned to the caller (ElevationGarage). The *PreCondition* ensures that the car with the associated LicensePlate is located within the domain, holding the available cars.

As an example of a *Recursive Function*, the getNumberOfParkedCars was added, it is only used in the VDMUnit test for the ParkingLevel, where it counts the number of cars in the ParkingLevel.

```
-- For testing purpose only:
public getNumberOfParkedCars : map LicensePlate to Car -> nat
getNumberOfParkedCars(pps) ==
   if pps = {|->}
```

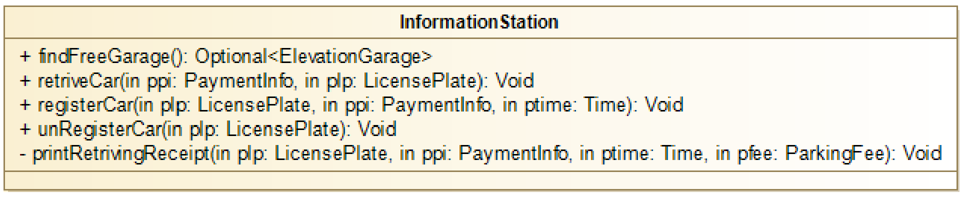
```
      then 0
    else
      let y in set dom pps
      in
      1 + getNumberOfParkedCars(({y} <-: pps))
measure Card;

Card : map LicensePlate to Car -> nat
Card(pps) ==
card dom pps
```

Listing 15 – getNumberOfParkedCars function

The function simply counts the number of cars by recursively calling itself. To ensure that the recursive function will terminate, the measure of the domain of the mapping is added.

4.6 InformationStation

The InformationStation is the first component that the user meets. It holds the responsibilities of finding a free garage for the user, the initiation of retrieval of a car, as well as withdrawing the payment from the user's payment information and providing a receipt for the user.



InformationStation

+ findFreeGarage(): Optional<ElevationGarage>
+ retriveCar(in ppi: PaymentInfo, in plp: LicensePlate): Void
+ registerCar(in plp: LicensePlate, in ppi: PaymentInfo, in ptime: Time): Void
+ unRegisterCar(in plp: LicensePlate): Void
- printRetrivingReceipt(in plp: LicensePlate, in ppi: PaymentInfo, in ptime: Time, in pfee: ParkingFee): Void

Figure 6 - InformationStation Class

It has also been decided that all registrations and unregistrations to the database must go through the InformationStation.

```
pure public findFreeGarage : () ==> [ElevationGarage]
findFreeGarage() == (
               let freeGarage in seq APS`garages be st freeGarage.isGarageFree()
               in
               return freeGarage;)
pre APS`garages <> [] and exists x in seq APS`garages & x.isGarageFree();
```

Listing 16 – findFreeGarage operation

The findFreeGarage operation is used, when the user wishes to park a car. The operation uses a *let-be-such-that* expression which returns an ElevationGarage that returns true on isGarageFree.

The *PreCondition* checks for content in the database, and ensures that a free garage is available.

```
public retriveCar : PaymentInfo * LicensePlate ==> ()
retriveCar(ppi, plp) == (
  dcl freeGarage : [ElevationGarage] := findFreeGarage();
  dcl parkingFee : ParkingFee;
  if freeGarage <> nil
     then if APS`validationService.validatePaymentInfo(ppi)
          then if card {x.license | x in set APS`garageDB & x.info = ppi} > 1
               then (let y in set APS`garageDB be st y.license = plp
                     in
                     (parkingFee := APS`transactionService.calculateParkingFee(plp, ppi, y.parkTime);
                      printRetrivingReceipt(plp, ppi, y.parkTime, parkingFee););
                      freeGarage.retriveCar(plp);
                      APS`io.print("STATUS: garageDB contains: ");APS`io.print(APS`garageDB);
                      APS`io.print("\n");)
               else
                   (let parkedCarInfo in set APS`garageDB be st parkedCarInfo.info = ppi
                    in
                   (freeGarage.retriveCar(parkedCarInfo.license);
                    parkingFee := APS`transactionService.calculateParkingFee(plp, ppi,
                                                                         parkedCarInfo.parkTime);
                    printRetrivingReceipt(plp, ppi, parkedCarInfo.parkTime, parkingFee););
                    APS`io.print("STATUS: garageDB contains: ");APS`io.print(APS`garageDB);
                    APS`io.print("\n"))
     else APS`io.print("STATUS: Error - <PaymentCardInvalid>\n")
  else APS`io.print("STATUS: Error - <NoFreeGarages>\n"););
pre APS`garageDB <> {};
```

Listing 17 – retriveCar operatio

The retriveCar operation is used by the user to retrieve a Car, by inserting the same PaymentInfo which the car was originally parked with. First the findFreeGarage presented above on [Listing 16 – findFreeGarage operation] is used, and if a freeGarage is available the system validates the payment card inserted by the user. If the payment is accepted, the system checks whether more than one car is parked using the payment information. This check is done by first using a *Set Comprehension* to find all elements in the database registered with the inserted paymentInformation, by then taking the cardinality using the *card* operator, it is checked if more than one licensePlate is registered. If so, a *let-be-such-that* expression is used to retrieve the car with the LicensePlate provided by the user. For the applicable car, the system now calculates the parkingFee using the TransactionService, by handing over the parking information. A receipt is printed for the user, and the retriveCar operation is then called on the freeGarage, which implicit has the user go to the freeGarage.

If the PaymentInformation only is used for parking a single car, another *let-be-such-that* expression is used, retrieving the car without the need of the LicensePlate.

```
public registerCar : LicensePlate * PaymentInfo * Time ==> ()
registerCar(plp, ppi, ptime) ==
   APS`garageDB := APS`garageDB union {mk_ParkedCarInfo(plp, ppi, ptime)}
pre mk_ParkedCarInfo(plp, ppi, ptime) not in set APS`garageDB;
```
Listing 18 – registerCar operation

The registration and unregistration of elements to and from the database is also handled by the InformationStation. As the garageDB is a set, the registering is done using the *union* operator and the unregistering is done using a *Set Comprehension* identifying the desired Car using it's LicensePlate and having it removed by the difference \ operator.

The *Pre Condition* of the registerCar ensures that the car does not already exist by using the *mk_* operator of the information to be inserted in the database.

```
public unRegisterCar : LicensePlate ==> ()
unRegisterCar(plp) ==
  APS`garageDB := APS`garageDB \ { parkedCarInfo|
                parkedCarInfo in set APS`garageDB & parkedCarInfo.license = plp}
pre exists x in set APS`garageDB & x.license = plp;
```
Listing 19 – unRegisterCar operation

Lastly the InformationStation is responsible of creating output when a car is being retrieved from the station. This is done using the IO library to write out a receipt in the console for the user.

```
private printRetrivingReceipt : LicensePlate * PaymentInfo * Time * ParkingFee ==> ()
printRetrivingReceipt(plp, ppi, ptime, pfee) == (
   APS`io.print("---------------------\n");
   APS`io.print("  Receiving Receipt  \n");
   APS`io.print("Car: "); APS`io.print(plp); APS`io.print("\n");
   APS`io.print("Parked at time: "); APS`io.print(ptime); APS`io.print("\n");
   APS`io.print("Current time: "); APS`io.print(World`timeRef.getTime()); APS`io.print("\n");
   APS`io.print("Payment method: ");
   cases is_ParkingCard(ppi):
      true -> APS`io.print("Parking Card\n"),
      false -> APS`io.print("Credit Card\n")
   end;
   APS`io.print("Payment fee: "); APS`io.print(pfee); APS`io.print("\n");
   APS`io.print("---------------------\n"););
```
Listing 20 – printRetrivingReceipt operation

The operation uses a *Cases Expression* to determine the type of the payment method and print the right string depending on the result. This could also be done by using an if-then-else expression, but to show the use of Cases, it was chosen as the concept.

4.7 TransactionService

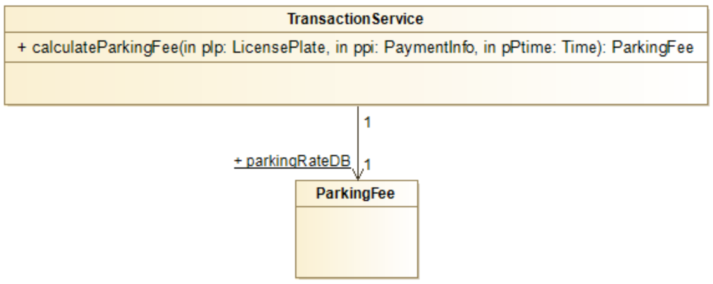

TransactionService
+ calculateParkingFee(in plp: LicensePlate, in ppi: PaymentInfo, in pPtime: Time): ParkingFee
1
+ parkingRateDB
1
ParkingFee

Figure 7 – TransactionService Class

The TransactionService is used by the InformationStation to calculate the parking fee to be withdrawn.

```
values
public parkingRateDB : map CountryIdentifier to ParkingFee
                       =  {<Danish>   |-> 5,
                           <European> |-> 10,
                           <Other>    |-> 15};
```

Listing 21 – parkingRateDB mapping

The TransactionService holds a value of the type a *mapping*. The mapping is from the domain CountryIdentifier holded by a LicensePlate to the range of ParkingFee. This is used as a rule set of the parking rates.

```
public calculateParkingFee : LicensePlate * PaymentInfo * Time ==> ParkingFee
calculateParkingFee(plp, ppi, pPtime) ==
    if is_ParkingCard(ppi)
       then return 0
    else return parkingRateDB(plp.ci) * (World`timeRef.getTime() - pPtime)
pre pPtime > 0 and plp.ci in set dom parkingRateDB
post is_(RESULT,nat);
```

Listing 22 – calculateParkingFee operation

The calculation of parking fee is done by first checking if the provided Payment method is of the type ParkingCard. If so, the fee should be of the value 0, since this is considered as a subscription system, where the user does not pay for the specific parking. This is done using an *if-then-else* expression which on the else acts by calculating the parking fee by returning the parkingRate of the associated car nationality times the time that the car was parked. The current time is gathered using the getTime operation of the Timer class, and the parked time is provided when the user parks a car.

4.8 Environment

To stimulate the system with user inputs, an Environment Class was created. The more a system depends on user inputs, the more abstract the interaction often gets.

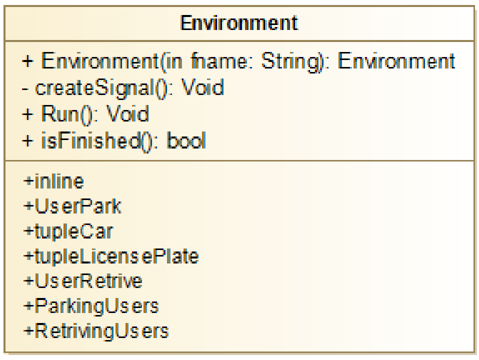


Figure 8 - Environment Class

In this system, the input to the system is seen as sequences of users who wishes to park a car and users who wishes to retrieve already parked cars.

```
Types
protected inline = (seq of UserPark * seq of UserRetrive);

private UserPark = (Time * PaymentInfo * tupleCar);
private UserRetrive = (Time * PaymentInfo * tupleLicensePlate);
```

Listing 23 – Environment types

As indicated by the types of the Environment class, the types read from the input file needs to be tuples. Therefore tupleCar and tupleLicensePlate is created, holding the same information as the records used in the GLOBAL class.

```
instance variables
  io   : IO   := new IO();
  busy : bool := true;
  parkingUsers   : seq of ParkingUsers   := [];
  retrivingUsers : seq of RetrivingUsers := [];
```

Listing 24 – Environment instance variables

The Environment class instance variables consists of a IO to provide the possibility of writing when needed to the console, a Boolean type identifying if the system is busy reading input and two sequences of users as described.

```
public Environment : String ==> Environment
Environment(fname) ==
def mk_(-,mk_(userPark, userRet)) = io.freadval[inline](fname)
in (parkingUsers := [mk_ParkingUsers(x.#1, x.#2, mk_Car(mk_LicensePlate(x.#3.#1.#1, x.#3.#1.#2,
                                                    x.#3.#1.#3), x.#3.#2)) | x in seq userPark];

    retrivingUsers := [mk_RetrivingUsers(x.#1, x.#2, mk_LicensePlate(x.#3.#1, x.#3.#2, x.#3.#3))
                                                                      | x in seq userRet];);
```

Listing 25 – Environment constructor

The constructor of the Environment reads a scenario text file which defines the input stimulus to the system. By using a *Sequence Comprehension* which returns the desired records retrieved through the inline the sequences of users is constructed.

```
private createSignal : () ==> ()
createSignal() ==
  (if len parkingUsers > 0 or len retrivingUsers > 0
   then
       (dcl
           currentTime   : Time := World`timeRef.getTime(),
           parkingDone   : bool := false,
           retrivingDone : bool := false;
       busy := true;
       while not parkingDone and not retrivingDone
             do
                (if len parkingUsers > 0
                 then
                     (let
                         ParkingUser = hd parkingUsers
                      in
                         if ParkingUser.time <= currentTime
                         then
                             (dcl ElevGarage : ElevationGarage := APS`informationStation.findFreeGarage();
                              ElevGarage.parkCar(ParkingUser.car);
                              ElevGarage.handleParking(ParkingUser.payInfo);
                              parkingUsers := tl parkingUsers;
                              parkingDone := len parkingUsers = 0
                              )
                        else parkingDone := true)
                 else parkingDone := true;
                 if len retrivingUsers > 0
                 then
                     (let
                         RetrivingUser = hd retrivingUsers
                      in
                         if RetrivingUser.time <= currentTime
                         then
                          (APS`informationStation.retriveCar(RetrivingUser.payInfo,RetrivingUser.licPlate);
                           retrivingUsers := tl retrivingUsers;
                           retrivingDone := len retrivingUsers = 0)
                        else retrivingDone := true
                else retrivingDone := true))
  else busy := false)
pre parkingUsers <> [] => retrivingUsers <> []
```

Listing 26 – Enviroment createSignal

By creating a signal, the execution of the system in order to meet the input stimuli is handled. The signal goes through every element in the two input sequences, when either a ParkingUser or a RetrivingUser with the current time arrives it is handled. When every tuple in the file is handled, the busy is changed to false, and the system execution ends.

5 Results

To ensure the validation and verification of the system, it is tested using both scenarios, to test the complete system, and VDMUnit, to ensure the functionality of the individual classes.

5.1 Scenarios

During the testing, multiple scenarios has been created to ensure that the complete system is working as expected and returns the right output.

```
mk_(
   [
   mk_(1,[9,0,9,0,0,1],mk_(mk_(<Danish>, "PT", 49000),false)),
   mk_(3,[9,0,9,0,0,1],mk_(mk_(<Other>, "KS", 49001),false)),
   mk_(7,['p','4','5','1','0','0'],mk_(mk_(<European>, "PT", 98042),true))
   ],
   [
   mk_(8,[9,0,9,0,0,1],mk_(<Danish>, "PT", 49000)),
   mk_(9,['p','4','5','1','0','0'],mk_(<European>, "PT", 98042)),
   mk_(10,[9,0,9,0,0,1],mk_(<Other>, "KS", 49001))
])
```

Listing 27 – scenario.txt

The listing showed above ensures a scenario fulfilling the key functionalities of the system:

- A user parking multiple cars using the same payment information.
- A user parking a car containing a parking permit using a parking card.
- Retrieval of the cars one by one.
- Correct payments both for normal credit card and parking card.

The expected output is shown below, where a combination of Receipts and system STATUS prints validates the correctness of the system.

```
**
** Overture Console
**
STATUS: Step
---------------------
  Parking Receipt
Car: mk_Car(mk_LicensePlate(<Danish>, "PT", 49000), false)
Current time: 1
Payment method: Credit Card
---------------------
STATUS: garageDB contains: {mk_ParkedCarInfo(mk_LicensePlate(<Danish>, "PT", 49000), [9, 0, 9, 0, 0, 1], 1)}
STATUS: Step
STATUS: Step
---------------------
  Parking Receipt
Car: mk_Car(mk_LicensePlate(<Other>, "KS", 49001), false)
Current time: 3
Payment method: Credit Card
---------------------
STATUS: garageDB contains: {mk_ParkedCarInfo(mk_LicensePlate(<Danish>, "PT", 49000), [9, 0, 9, 0, 0, 1], 1),
mk_ParkedCarInfo(mk_LicensePlate(<Other>, "KS", 49001), [9, 0, 9, 0, 0, 1], 3)}
STATUS: Step
STATUS: Step
STATUS: Step
STATUS: Step
---------------------
  Parking Receipt
Car: mk_Car(mk_LicensePlate(<European>, "PT", 98042), true)
Current time: 7
Payment method: Parking Card
---------------------
STATUS: garageDB contains: {mk_ParkedCarInfo(mk_LicensePlate(<Danish>, "PT", 49000), [9, 0, 9, 0, 0, 1], 1),
mk_ParkedCarInfo(mk_LicensePlate(<European>, "PT", 98042), "p45100", 7), mk_ParkedCarInfo(mk_LicensePlate(<Other>, "KS",
49001), [9, 0, 9, 0, 0, 1], 3)}
```

```
STATUS: Step
---------------------
  Receiving Receipt
Car: mk_LicensePlate(<Danish>, "PT", 49000)
Parked at time: 1
Current time: 8
Payment method: Credit Card
Payment fee: 35
---------------------
STATUS: garageDB contains: {mk_ParkedCarInfo(mk_LicensePlate(<European>, "PT", 98042), "p45100", 7),
mk_ParkedCarInfo(mk_LicensePlate(<Other>, "KS", 49001), [9, 0, 9, 0, 0, 1], 3)}
STATUS: Step
---------------------
  Receiving Receipt
Car: mk_LicensePlate(<European>, "PT", 98042)
Parked at time: 7
Current time: 9
Payment method: Parking Card
Payment fee: 0
---------------------
STATUS: garageDB contains: {mk_ParkedCarInfo(mk_LicensePlate(<Other>, "KS", 49001), [9, 0, 9, 0, 0, 1], 3)}
STATUS: Step
---------------------
  Receiving Receipt
Car: mk_LicensePlate(<Other>, "KS", 49001)
Parked at time: 3
Current time: 10
Payment method: Credit Card
Payment fee: 105
---------------------
STATUS: garageDB contains: {}
STATUS: Step
STATUS: Step
```

Listing 28 – console output

5.2 VDMUnit Test

As well as scenarios, the system is tested using VDMUnit by including the VDMUnit library. The five system classes ParkingLevel, ValidationService, TransactionService, ElevationGarage and InformationStation are all tested.

```
class APSTest
operations

public Execute : () ==> ()
Execute() ==
  let ts : TestSuite = new TestSuite(),
  Testresult = new TestResult()
  in
    (ts.addTest(new ParkingLevelTest());
     ts.addTest(new ValidationServiceTest());
     ts.addTest(new TransactionServiceTest());
     ts.addTest(new ElevationGarageTest());
     ts.addTest(new InformationStationTest());
     ts.run(Testresult);
     IO`println(Testresult.toString()););
end APSTest
```

Listing 29 – APSTest

A common entry point for all the five tests is created using the APSTest class, which creates a TestSuite which is started on execution of the class. The test classes responding to the five system classes is then added to the suite, so that they are executed.

An example of a single unit test is shown in the following listing, where the TransactionService's operation calculateParkingFee is tested:

```
class TransactionServiceTest is subclass of TestCase, GLOBAL

values
car1 : Car = mk_Car(mk_LicensePlate(<Danish>, "PT", 49000), true);

payInfo_cred1 : PaymentInfo = [1,2,3,4,5,6];
payInfo_park1 : PaymentInfo = ['p','0','0','0','0','1'];

parkingTime : real = 1;

operations
protected runTest : () ==> ()
runTest() ==
   (calculateParkingFeeTest();
    parkingRateDBTest());

…

protected calculateParkingFeeTest : () ==> ()
calculateParkingFeeTest() == (
dcl transactionService : TransactionService := new TransactionService();
World`timeRef.stepTime();
World`timeRef.stepTime();
World`timeRef.stepTime();
World`timeRef.stepTime();
World`timeRef.stepTime();
assertTrue(transactionService.calculateParkingFee(car1.licensePlate, payInfo_cred1, parkingTime) = 20);
assertFalse(transactionService.calculateParkingFee(car1.licensePlate, payInfo_park1, parkingTime) <> 0););

end TransactionServiceTest
```

Listing 30 – TransactionServiceTest

First an instance of a Car is created with a Danish nationality in addition of two PaymentInfo's, a credit card and a parking card. Also a time is provided, to test the functionality of the operation. The runTest operation is handling the execution of all nested tested within the class. The calculation of the parking fee is tested by creating a new instance of the TransactionService, and stepping the time five times. At the current time, hence five, the calculateParkingFee operation is called, first with a credit card, which results in a price of:

$$P_{fee} = P_{rate} * (t_{current} - t_{park}) = 5 * (5 - 1) = 20$$

Which is validated by using the assertTrue, which assumes a true statement. Finally the same operation is run with a ParkingCard type, and using the car which has a ParkingPermit. The assertFalse is used here for the statement providing another result than 0.

```
----------------------------------------
|    TEST RESULTS                      |
|--------------------------------------|
| Executed: 5
| Failures: 0
| Errors:   0
|______________________________________|
|
|
|--------------------------------------|
|               SUCCESS                |
|______________________________________|

new APSTest().Execute() = ()
```

Listing 31 – Console output of APSTest().Execute()

6 Future work

The developed system is implemented as a Sequential Design Model. Of course a real life Automated Parking System would include many non-sequential and therefore concurrent aspects. To meet these aspects, the model should be expanded to be a Concurrent Design Model.

The major change in converting a model to be concurrent is the introduction of threads. This introduction is followed up with handling the communication between these individual threads, and ensuring that the model works without any locks, collisions or other dilemmas introduces by having multiple threads. Ensuring of this synchronization between the threads is acquired using *Permission Predicates* or *Mutex'* which can be seen as guards or locks ensuring the flow of the system.

In the Automated Parking System, the first thing that should be considered is the shared resources, such as the ParkingLevel and the garageDB. As the ParkingLevel should be accessible for multiple concurrently executed ElevationGarages, Permission Predicates would be a valid tool to ensure who could access this media at which times. The same case is considered for the database, where readings and writings to it can happen in multiple places in the system.

Some elements in the current sequential model would prove very functional in such a scenario, for instance the getFreeGarage operation which then would have to actually choose a garage between multiple active garages. Also the gate handling of each ElevationGarage could be used in cooperation with a Mutex to ensure no user is accessing a busy garage.

As a final step, the model could be implemented using VDM-RT, which provides the possibility to model a system as a real-time model. This would allow for the possibility of determine deadlines in the system which should be respected to meet the systems specification and requirements. Also allocation of busses and processing units is possible in VDM-RT providing a deterministic model.

7 Conclusion

The project has resulted in a working model of an Automated Parking System. It is possible for a sequence of users to park one or more cars containing unique license plates. The payment of the parking can be done through two payment methods, which results in different parking fees. When a car is parked into the system, its license plate is first validated to ensure only valid vehicles are parked in the parking level. The parked cars' license plates is added to a database, holding the information needed. Sequence of users who wishes to retrieve prior parked cars can do so, and choose which car to retrieve if more than one is parked using the same payment information. When a car is retrieved the payment fee of the parking is calculated and an output in form of a receipt is printed for the user.

The developed model is validated using both VDMUnit tests and scenarios testing the system as a whole. The collection of requirements listed in the beginning of the report [2 Requirements] is considered meet, and the system is working as expected.

The case work of an Automated Parking System has turned out as a great foundation for modeling of mission critical systems using VDM++. It has allowed for the use of many VDM concepts, and contains mission critical aspects which is needed to be executed correctly to ensure user safety.

AutomatedParkingSystem

May 23, 2017

Contents

1 APS

```
class APS is subclass of GLOBAL

instance variables
 public static eg1 : ElevationGarage := new ElevationGarage();
 public static eg2 : ElevationGarage := new ElevationGarage();

 public static garages : seq of ElevationGarage := [eg1, eg2];

 public static informationStation : InformationStation := new InformationStation();

 public static parkingLevel : ParkingLevel := new ParkingLevel();

 public static validationService : ValidationService := new ValidationService();

 public static transactionService : TransactionService := new TransactionService();

 public static garageDB : set of ParkedCarInfo := {};

 public static io : IO := new IO();
end APS
```

Function or operation	Line	Coverage	Calls
APS.vdmpp		100.0%	0

2 ElevationGarage

```
class ElevationGarage is subclass of GLOBAL

types

public GateState = <Open> | <Closed>;

instance variables
 EntranceGate : GateState := <Open>;
 ExitGate : GateState := <Closed>;
 parkedCar : [Car] := nil;
 isFree : bool := true;

operations


public parkCar : Car ==> ()
parkCar(pcar) == (
 parkedCar := pcar;
 isFree := false;)
pre parkedCar = nil;


public handleParking:  PaymentInfo ==> ()
handleParking(ppi) == (
 if not isFree
  then (
     if APS`validationService.validateLicensePlate(parkedCar.licensePlate) and
     APS`validationService.validatePaymentInfo(ppi,parkedCar.parkingPermit)
       then (garageBusy();
          APS`informationStation.registerCar(parkedCar.licensePlate,ppi,World`timeRef.getTime());
          APS`parkingLevel.parkCar(parkedCar);
          printParkingReceipt(ppi);
          APS`io.print("STATUS: garageDB contains: ");APS`io.print(APS`garageDB); APS`io.print("\
              n");
          parkedCar := nil;
          garageFree();)
     else
     (garageReadyForExit();
     APS`io.print("STATUS: Error - <ValidationFailed>\n");
     parkedCar := nil;
     garageFree();)
     )
)
pre EntranceGate = <Open> and ExitGate = <Closed> and isFree = false;


public retriveCar: LicensePlate ==> ()
retriveCar(plp) == (
 isFree := false;
 garageBusy();
 parkedCar := APS`parkingLevel.retriveCar(plp);
 garageReadyForExit();
 parkedCar := nil;
 APS`informationStation.unRegisterCar(plp);
 garageFree();
 isFree := true)
pre EntranceGate = <Open> and ExitGate = <Closed> and
  isFree = true and APS`garageDB <> {} ;


pure public isGarageFree: () ==> bool
isGarageFree() ==
 return isFree;


public garageFree: () ==> ()
garageFree() == (
 if EntranceGate = <Closed>
   then EntranceGate := <Open>;
 if ExitGate = <Open>
   then ExitGate := <Closed>;
 isFree := true;)
pre EntranceGate = ExitGate and isFree = false;


public garageBusy: () ==> ()
garageBusy() == (
 EntranceGate := <Closed>;)
pre EntranceGate = <Open> and ExitGate = <Closed> and isFree = false;


public garageReadyForExit: () ==> ()
garageReadyForExit() == (
 if EntranceGate = <Closed>
   then EntranceGate := <Open>;
 ExitGate := <Open>;)
pre ExitGate = <Closed> and isFree = false;


public getGateStates: () ==> seq of GateState
getGateStates() ==
 return [EntranceGate, ExitGate];


private printParkingReceipt : PaymentInfo ==> ()
printParkingReceipt(ppi) == (
  APS`io.print("--------------------\n");
  APS`io.print("  Parking Receipt  \n");
  APS`io.print("Car: "); APS`io.print(parkedCar); APS`io.print("\n");
  APS`io.print("Current time: "); APS`io.print(World`timeRef.getTime()); APS`io.print("\n");
  APS`io.print("Payment method: ");
  cases is_ParkingCard(ppi):
   true -> APS`io.print("Parking Card\n"),
   false -> APS`io.print("Credit Card\n")
  end;
  APS`io.print("---------------------\n"););

end ElevationGarage
```

Function or operation	Line	Coverage	Calls
garageBusy	69	100.0%	14
garageFree	60	100.0%	14
garageReadyForExit	74	100.0%	7
getGateStates	81	100.0%	4
handleParking	21	87.2%	6
isGarageFree	56	100.0%	26
parkCar	15	100.0%	8
printParkingReceipt	85	100.0%	6
retriveCar	43	100.0%	6
ElevationGarage.vdmpp		96.2%	91

3 InformationStation

```
class InformationStation is subclass of GLOBAL

operations


pure public findFreeGarage : () ==> [ElevationGarage]
findFreeGarage() == (
 let freeGarage in seq APS`garages be st freeGarage.isGarageFree()
 in
  return freeGarage;)
pre APS`garages <> [] and exists x in seq APS`garages & x.isGarageFree();


public retriveCar : PaymentInfo * LicensePlate ==> ()
retriveCar(ppi, plp) == (
 dcl freeGarage : [ElevationGarage] := findFreeGarage();
 dcl parkingFee : ParkingFee;

 if freeGarage <> nil

  then if APS`validationService.validatePaymentInfo(ppi)
      then if card {x.license | x in set APS`garageDB & x.info = ppi} > 1

        then (let y in set APS`garageDB be st y.license = plp
        in
        (parkingFee := APS`transactionService.calculateParkingFee(plp, ppi, y.parkTime);
         printRetrivingReceipt(plp, ppi, y.parkTime, parkingFee););
        freeGarage.retriveCar(plp);
        APS`io.print("STATUS: garageDB contains: ");APS`io.print(APS`garageDB); APS`io.print("\n"
            );)
      else

        (let parkedCarInfo in set APS`garageDB be st parkedCarInfo.info = ppi
        in (
        freeGarage.retriveCar(parkedCarInfo.license);
        parkingFee := APS`transactionService.calculateParkingFee(plp, ppi, parkedCarInfo.parkTime
            );
        printRetrivingReceipt(plp, ppi, parkedCarInfo.parkTime, parkingFee););
        APS`io.print("STATUS: garageDB contains: ");APS`io.print(APS`garageDB); APS`io.print("\n"
            ))

  else APS`io.print("STATUS: Error - <PaymentCardInvalid>\n")
else APS`io.print("STATUS: Error - <NoFreeGarages>\n");)
pre APS`garageDB <> {};


public registerCar : LicensePlate * PaymentInfo * Time ==> ()
registerCar(plp, ppi, ptime) ==
 APS`garageDB := APS`garageDB union {mk_ParkedCarInfo(plp, ppi, ptime)}
pre mk_ParkedCarInfo(plp, ppi, ptime) not in set APS`garageDB;


public unRegisterCar : LicensePlate ==> ()
unRegisterCar(plp) ==
 APS`garageDB := APS`garageDB \ { parkedCarInfo|
 parkedCarInfo in set APS`garageDB & parkedCarInfo.license = plp}
pre exists x in set APS`garageDB & x.license = plp;


private printRetrivingReceipt : LicensePlate * PaymentInfo * Time * ParkingFee ==> ()
printRetrivingReceipt(plp, ppi, ptime, pfee) == (
  APS`io.print("---------------------\n");
  APS`io.print("  Receiving Receipt  \n");
  APS`io.print("Car: "); APS`io.print(plp); APS`io.print("\n");
  APS`io.print("Parked at time: "); APS`io.print(ptime); APS`io.print("\n");
  APS`io.print("Current time: "); APS`io.print(World`timeRef.getTime()); APS`io.print("\n");
  APS`io.print("Payment method: ");
  cases is_ParkingCard(ppi):
   true -> APS`io.print("Parking Card\n"),
   false -> APS`io.print("Credit Card\n")
  end;
  APS`io.print("Payment fee: "); APS`io.print(pfee); APS`io.print("\n");
  APS`io.print("---------------------\n"););

end InformationStation
```

Function or operation	Line	Coverage	Calls
findFreeGarage	5	100.0%	12
printRetrivingReceipt	52	100.0%	6
registerCar	41	100.0%	8
retriveCar	12	95.4%	6
unRegisterCar	46	100.0%	8
InformationStation.vdmpp		97.7%	40

4 ParkingLevel

```
class ParkingLevel is subclass of GLOBAL
```

```
instance variables
parkingSpots : map LicensePlate to Car := {|->};
inv forall x,y in set rng parkingSpots & x <> y => x.licensePlate <> y.licensePlate

operations


public parkCar : Car ==> ()
parkCar(pcar) ==
 parkingSpots := parkingSpots munion {pcar.licensePlate |-> pcar}
pre forall i in set rng parkingSpots & pcar.licensePlate <> i.licensePlate;


public retriveCar : LicensePlate ==> Car
retriveCar(plp) == (
 dcl car : Car := parkingSpots(plp);
 parkingSpots := {plp} <-: parkingSpots;
 return car;)
pre plp in set dom parkingSpots;

-- For testing purpose only:

pure public getParkingSpotsMap : () ==> map LicensePlate to Car
getParkingSpotsMap() ==
  return  parkingSpots;

functions

-- For testing purpose only:

public getNumberOfParkedCars : map LicensePlate to Car -> nat
getNumberOfParkedCars(pps) ==
 if pps = {|->}
 then 0
 else
  let y in set dom pps
  in
   1 + getNumberOfParkedCars(({y} <-: pps))
measure Card;


Card : map LicensePlate to Car -> nat
Card(pps) ==
 card dom pps


end ParkingLevel
```

Function or operation	Line	Coverage	Calls
Card	39	100.0%	12
getNumberOfParkedCars	29	100.0%	12
getParkingSpotsMap	22	100.0%	5
parkCar	9	100.0%	12
retriveCar	14	100.0%	8
ParkingLevel.vdmpp		100.0%	49

5 TransactionService

```
class TransactionService is subclass of GLOBAL

values
public parkingRateDB : inmap CountryIdentifier to ParkingFee
     = {<Danish> |-> 5,
        <European> |-> 10,
       <Other> |-> 15};

operations


public calculateParkingFee : LicensePlate * PaymentInfo * Time ==> ParkingFee
calculateParkingFee(plp, ppi, pPtime) ==
 if is_ParkingCard(ppi)
   then return 0
 else return parkingRateDB(plp.ci) * (World`timeRef.getTime() - pPtime)
pre pPtime > 0 and plp.ci in set dom parkingRateDB
post is_(RESULT,nat);

end TransactionService
```

Function or operation	Line	Coverage	Calls
calculateParkingFee	11	100.0%	8
TransactionService.vdmpp		100.0%	8

6 ValidationService

```
class ValidationService is subclass of GLOBAL
-- Ref: https://da.wikipedia.org/wiki/Nummerplade
values

alphabet = {'A','B','C','D','E','F','G','H','J','K','L','M','N','O','P','R','S','T','U','V','X','
    Y','Z'};

bannedRegistrationCombinations = {"BH", "BU", "CC", "CD", "DK", "DU", "EU", "KZ", "MU", "PY", "SS
    ", "UD", "UN", "VC"};

operations


public validateLicensePlate : LicensePlate ==> bool
validateLicensePlate(plp) == (
 return validateRegistrationLetters(plp) and validateNumberSeries(plp) and
   plp not in set {x.license | x in set APS`garageDB};);

functions


public validateRegistrationLetters : LicensePlate -> bool
validateRegistrationLetters(plp) ==
  plp.rl not in set bannedRegistrationCombinations and
 plp.rl(2) <> 'O'
pre plp.rl(1) in set alphabet and plp.rl(2) in set alphabet;
```

```
public validateNumberSeries : LicensePlate -> bool
validateNumberSeries(plp) ==
 plp.ns >= 5000 and plp.ns <= 5399 or
 plp.ns >= 20000 and plp.ns <= 77999 or
 plp.ns >= 98000 and plp.ns <= 99999;


public validatePaymentInfo : PaymentInfo * bool -> bool -- For parking
validatePaymentInfo(ppi, pHasParkingPermit) ==
 if is_CreditCard(ppi)
   then validateCreditCard(ppi)
 else if is_ParkingCard(ppi) and pHasParkingPermit
   then validateParkingCard(ppi)
 else false
pre forall x in seq ppi & x <> ' ';

public validatePaymentInfo : PaymentInfo -> bool -- For retrival
validatePaymentInfo(ppi) ==
 if is_CreditCard(ppi)
   then validateCreditCard(ppi)
 else if is_ParkingCard(ppi)
   then validateParkingCard(ppi)
 else false;


private validateCreditCard : PaymentInfo -> bool
validateCreditCard(ppi) ==
 len ppi = 6
pre ppi <> [];


private validateParkingCard : PaymentInfo -> bool
validateParkingCard(ppi) ==
 len ppi = 6 and hd ppi = 'p'
pre ppi <> [];

end ValidationService
```

Function or operation	Line	Coverage	Calls
validateCreditCard	48	100.0%	11
validateLicensePlate	11	100.0%	7
validateNumberSeries	25	100.0%	18
validateParkingCard	53	100.0%	7
validatePaymentInfo	31	90.9%	9
validateRegistrationLetters	18	100.0%	18
ValidationService.vdmpp		99.3%	70

7 Environment

```
class Environment is subclass of GLOBAL

types
```

```
protected inline = (seq of UserPark * seq of UserRetrive);

private UserPark = (Time * PaymentInfo * tupleCar);
private UserRetrive = (Time * PaymentInfo * tupleLicensePlate);


private tupleCar = (tupleLicensePlate * bool);
private tupleLicensePlate = (CountryIdentifier * RegistrationLetters * NumberSeries);

public ParkingUsers ::
 time    : Time
 payInfo : PaymentInfo
 car     : Car;

public RetrivingUsers ::
 time    : Time
 payInfo : PaymentInfo
 licPlate: LicensePlate;

instance variables
  io : IO := new IO();
 busy : bool := true;

 parkingUsers   : seq of ParkingUsers := [];
 retrivingUsers : seq of RetrivingUsers := [];

operations

public Environment : String ==> Environment
Environment(fname) ==
 def mk_(-,mk_(userPark, userRet)) = io.freadval[inline](fname)

 in (parkingUsers := [mk_ParkingUsers(x.#1, x.#2, mk_Car(mk_LicensePlate(x.#3.#1.#1, x.#3.#1.#2,
     x.#3.#1.#3), x.#3.#2)) | x in seq userPark];
   retrivingUsers := [mk_RetrivingUsers(x.#1, x.#2, mk_LicensePlate(x.#3.#1, x.#3.#2, x.#3.#3)) |
        x in seq userRet];);



private createSignal : () ==> ()
createSignal() ==
 (if len parkingUsers > 0 or len retrivingUsers > 0
  then
    (dcl
      currentTime   : Time := World`timeRef.getTime(),
      parkingDone   : bool := false,
      retrivingDone : bool := false;
    busy := true;
    while not parkingDone and not retrivingDone
        do
          (if len parkingUsers > 0
           then
             (let
                 ParkingUser = hd parkingUsers
              in
                 if ParkingUser.time <= currentTime
                 then
                   (dcl ElevGarage : ElevationGarage := APS`informationStation.findFreeGarage();
                    ElevGarage.parkCar(ParkingUser.car);
                    ElevGarage.handleParking(ParkingUser.payInfo);
                    parkingUsers := tl parkingUsers;
                    parkingDone := len parkingUsers = 0
                    )
                 else parkingDone := true)
           else parkingDone := true;
           if len retrivingUsers > 0
           then
             (let
               RetrivingUser = hd retrivingUsers
              in
                if RetrivingUser.time <= currentTime
                then
                  (APS`informationStation.retriveCar(RetrivingUser.payInfo,RetrivingUser.licPlate
                      );
                   retrivingUsers := tl retrivingUsers;
                   retrivingDone := len retrivingUsers = 0)
                else retrivingDone := true)
           else retrivingDone := true))
  else busy := false)
pre parkingUsers <> [] => retrivingUsers <> [];

public Run : () ==> ()
Run() == (
  while not isFinished()
  do
   (createSignal();
    World`timeRef.stepTime();
    io.print("STATUS: Step\n")));


public isFinished : () ==> bool
isFinished() ==
 return parkingUsers = [] and retrivingUsers = [] and not busy;

end Environment
```

Function or operation	Line	Coverage	Calls
Environment	36	100.0%	2
Run	116	100.0%	2
createSignal	40	100.0%	24
isFinished	89	100.0%	26
Environment.vdmpp		100.0%	54

8 GLOBAL

```
class GLOBAL
types

-- General types --

public String = seq of char;

public Time = nat;

-- Car  --
public CountryIdentifier = <Danish> | <European> | <Other>;

public RegistrationLetters = String
inv rl == len rl = 2;
```

```
public NumberSeries = nat
inv num == num <= 99999 and num >= 5000;

public LicensePlate :: ci : CountryIdentifier
                  rl : RegistrationLetters
                  ns : NumberSeries;

public Car :: licensePlate : LicensePlate
       parkingPermit : bool;

-- PaymentInfo --

public CreditCard = seq of nat
inv num == len num = 6 and forall x in seq num & x <= 9;

public ParkingCard = String
inv pc == len pc = 6;

public PaymentInfo = CreditCard | ParkingCard;

public ParkingFee = nat;

-- garageDB --

public ParkedCarInfo ::
 license  : LicensePlate
 info    : PaymentInfo
 parkTime : Time;

---

end GLOBAL
```

Function or operation	Line	Coverage	Calls
GLOBAL.vdmpp		100.0%	0

9 APSTest

```
class APSTest
operations


public Execute : () ==> ()
Execute() ==
 let ts : TestSuite = new TestSuite(),
 Testresult = new TestResult()
 in
  (ts.addTest(new ParkingLevelTest());
  ts.addTest(new ValidationServiceTest());
  ts.addTest(new TransactionServiceTest());
  ts.addTest(new ElevationGarageTest());
  ts.addTest(new InformationStationTest());
  ts.run(Testresult);
  IO`println(Testresult.toString()););
end APSTest
```

Function or operation	Line	Coverage	Calls
Execute	4	100.0%	1
APSTest.vdmpp		100.0%	1

10 ElevationGarageTest

```
class ElevationGarageTest is subclass of TestCase, GLOBAL

values
car1 : Car = mk_Car(mk_LicensePlate(<Danish>, "PT", 49000), true);

operations


protected runTest : () ==> ()
runTest() ==
 (gateTest1();
  gateTest2(););


protected gateTest1 : () ==> ()
gateTest1() == (
 dcl elevationGarage : ElevationGarage := new ElevationGarage();
 elevationGarage.parkCar(car1);
 assertTrue(elevationGarage.isGarageFree() = false);
 elevationGarage.garageBusy();
 assertTrue(elevationGarage.getGateStates() = [<Closed>, <Closed>]);
 elevationGarage.garageFree();
 assertTrue(elevationGarage.isGarageFree() = true);
 assertTrue(elevationGarage.getGateStates() = [<Open>, <Closed>]););


protected gateTest2 : () ==> ()
gateTest2() == (
 dcl elevationGarage : ElevationGarage := new ElevationGarage();
 elevationGarage.parkCar(car1);
 elevationGarage.garageBusy();
 elevationGarage.garageReadyForExit();
 assertTrue(elevationGarage.getGateStates() = [<Open>, <Open>]);
 elevationGarage.garageFree();
 assertTrue(elevationGarage.getGateStates() = [<Open>, <Closed>]););

end ElevationGarageTest
```

Function or operation	Line	Coverage	Calls
gateTest1	13	100.0%	1
gateTest2	24	100.0%	1
runTest	8	100.0%	1
ElevationGarageTest.vdmpp		100.0%	3

11 InformationStationTest

```
class InformationStationTest is subclass of TestCase, GLOBAL

values

car1 : Car = mk_Car(mk_LicensePlate(<Danish>, "PT", 49000), true);
car2 : Car = mk_Car(mk_LicensePlate(<Other>, "GT", 49001), true);

parkingTime1 : real = 1;
parkingTime2 : real = 5;

payInfo_cred1 : PaymentInfo = [1,2,3,4,5,6];
payInfo_park1 : PaymentInfo = ['p','0','0','0','0','1'];

dummyDB : set of ParkedCarInfo = {mk_ParkedCarInfo(car1.licensePlate,payInfo_cred1,parkingTime1),
                 mk_ParkedCarInfo(car2.licensePlate,payInfo_park1,parkingTime2)};

operations


protected runTest : () ==> ()
runTest() ==
 (registerAndUnregisterCarTest(););


protected registerAndUnregisterCarTest : () ==> ()
registerAndUnregisterCarTest() == (
 dcl informationStation : InformationStation := new InformationStation();
 -- RegisterTest:
 assertTrue(APS`garageDB = {});
 informationStation.registerCar(car1.licensePlate,payInfo_cred1,parkingTime1);
 assertTrue(APS`garageDB = {mk_ParkedCarInfo(car1.licensePlate,payInfo_cred1,parkingTime1)});
 informationStation.registerCar(car2.licensePlate,payInfo_park1,parkingTime2);
 assertTrue(APS`garageDB = dummyDB);

 -- UnregisterTest:
 informationStation.unRegisterCar(car1.licensePlate);
 assertTrue(APS`garageDB = {mk_ParkedCarInfo(car2.licensePlate,payInfo_park1,parkingTime2)});
 informationStation.unRegisterCar(car2.licensePlate);
 assertTrue(APS`garageDB = {}););

end InformationStationTest
```

Function or operation	Line	Coverage	Calls
registerAndUnregisterCarTest	23	100.0%	1
runTest	19	100.0%	1
InformationStationTest.vdmpp		100.0%	2

12 ParkingLevelTest

```
class ParkingLevelTest is subclass of TestCase, GLOBAL

values
car1 : Car = mk_Car(mk_LicensePlate(<Danish>, "PT", 49000), true);
car2 : Car = mk_Car(mk_LicensePlate(<Danish>, "PT", 49001), true);
operations
```

```
protected runTest : () ==> ()
runTest() ==
 (parkCarTest();
  retriveCarTest();
  GetNumberOfParkedCarsTest(););


protected parkCarTest : () ==> ()
parkCarTest() == (
  dcl parkingLevel : ParkingLevel := new ParkingLevel();
  parkingLevel.parkCar(car1);
  assertTrue(parkingLevel.getNumberOfParkedCars(parkingLevel.getParkingSpotsMap()) = 1);
  parkingLevel.parkCar(car2);
 assertTrue(parkingLevel.getNumberOfParkedCars(parkingLevel.getParkingSpotsMap()) = 2););


protected retriveCarTest : () ==> ()
retriveCarTest() == (
 dcl parkingLevel : ParkingLevel := new ParkingLevel();
 parkingLevel.parkCar(car1);
 parkingLevel.parkCar(car2);
 assertTrue(parkingLevel.getNumberOfParkedCars(parkingLevel.getParkingSpotsMap()) = 2);
 assertTrue(parkingLevel.retriveCar(car1.licensePlate) = car1);
 assertTrue(parkingLevel.retriveCar(car2.licensePlate) = car2);
 assertTrue(parkingLevel.getNumberOfParkedCars(parkingLevel.getParkingSpotsMap()) = 0););


protected GetNumberOfParkedCarsTest : () ==> ()
GetNumberOfParkedCarsTest() == (
 dcl parkingLevel : ParkingLevel := new ParkingLevel();
 parkingLevel.parkCar(car1);
 parkingLevel.parkCar(car2);
 assertTrue(parkingLevel.getNumberOfParkedCars(parkingLevel.getParkingSpotsMap()) = 2));

end ParkingLevelTest
```

Function or operation	Line	Coverage	Calls
GetNumberOfParkedCarsTest	32	100.0%	1
parkCarTest	14	100.0%	1
retriveCarTest	22	100.0%	1
runTest	8	100.0%	1
ParkingLevelTest.vdmpp		100.0%	4

13 TransactionServiceTest

```
class TransactionServiceTest is subclass of TestCase, GLOBAL

values
car1 : Car = mk_Car(mk_LicensePlate(<Danish>, "PT", 49000), true);

payInfo_cred1 : PaymentInfo = [1,2,3,4,5,6];
payInfo_park1 : PaymentInfo = ['p','0','0','0','0','1'];

parkingTime : real = 1;
```

```
operations

protected runTest : () ==> ()
runTest() ==
 (calculateParkingFeeTest();
  parkingRateDBTest());


protected parkingRateDBTest : () ==> ()
parkingRateDBTest() == (
 dcl transactionService : TransactionService := new TransactionService();
 assertTrue(transactionService.parkingRateDB(<Danish>) = 5);
 assertTrue(transactionService.parkingRateDB(<European>) = 10);
 assertTrue(transactionService.parkingRateDB(<Other>) = 15););


protected calculateParkingFeeTest : () ==> ()
calculateParkingFeeTest() == (
 dcl transactionService : TransactionService := new TransactionService();
 World`timeRef.stepTime();
 World`timeRef.stepTime();
 World`timeRef.stepTime();
 World`timeRef.stepTime();
 World`timeRef.stepTime();
 assertTrue(transactionService.calculateParkingFee(car1.licensePlate, payInfo_cred1, parkingTime)
       = 20);
 assertFalse(transactionService.calculateParkingFee(car1.licensePlate, payInfo_park1, parkingTime
      ) <> 0););

end TransactionServiceTest
```

Function or operation	Line	Coverage	Calls
calculateParkingFeeTest	24	100.0%	1
parkingRateDBTest	17	100.0%	1
runTest	12	100.0%	1
TransactionServiceTest.vdmpp		100.0%	3

14 ValidationServiceTest

```
class ValidationServiceTest is subclass of TestCase, GLOBAL

values
car1 : Car = mk_Car(mk_LicensePlate(<Danish>, "GF", 49000), true);
car2 : Car = mk_Car(mk_LicensePlate(<Danish>, "BH", 10000), true);

payInfo_cred1 : PaymentInfo = [1,2,3,4,5,6];
payInfo_cred2 : PaymentInfo = [0,0,0,0,0,0];

payInfo_park1 : PaymentInfo = ['p','0','0','0','0','0'];
payInfo_park2 : PaymentInfo = ['p','1','2','3','4','5'];
payInfo_park3 : PaymentInfo = ['j','1','2','3','4','5'];

operations


protected runTest : () ==> ()
```

```
runTest() ==
 (validateLicensePlateTest();
  validateRegistrationLettersTest();
  validateNumberSeriesTest();
  validatePaymentInfoTest(););


protected validateLicensePlateTest : () ==> ()
validateLicensePlateTest() == (
 dcl validationService : ValidationService := new ValidationService();
 assertTrue(validationService.validateLicensePlate(car1.licensePlate) = true););


protected validateRegistrationLettersTest : () ==> ()
validateRegistrationLettersTest() == (
 dcl validationService : ValidationService := new ValidationService();
 assertTrue(validationService.validateRegistrationLetters(car1.licensePlate) = true);
 assertTrue(validationService.validateRegistrationLetters(car2.licensePlate) = false););


protected validateNumberSeriesTest : () ==> ()
validateNumberSeriesTest() == (
 dcl validationService : ValidationService := new ValidationService();
 assertTrue(validationService.validateNumberSeries(car1.licensePlate) = true);
 assertTrue(validationService.validateNumberSeries(car2.licensePlate) = false););


protected validatePaymentInfoTest : () ==> ()
validatePaymentInfoTest() == (
 dcl validationService : ValidationService := new ValidationService();
 assertTrue(validationService.validatePaymentInfo(payInfo_cred1) = true);
 assertTrue(validationService.validatePaymentInfo(payInfo_cred2) = true);
 assertTrue(validationService.validatePaymentInfo(payInfo_cred1, true) = true);
 assertTrue(validationService.validatePaymentInfo(payInfo_park1, true) = true);
 assertTrue(validationService.validatePaymentInfo(payInfo_park1) = true);
 assertTrue(validationService.validatePaymentInfo(payInfo_park2, false) = false);
 assertTrue(validationService.validatePaymentInfo(payInfo_park3, true) = false););

end ValidationServiceTest
```

Function or operation	Line	Coverage	Calls
runTest	16	100.0%	1
validateLicensePlateTest	23	100.0%	1
validateNumberSeriesTest	34	100.0%	1
validatePaymentInfoTest	40	100.0%	1
validateRegistrationLettersTest	28	100.0%	1
ValidationServiceTest.vdmpp		100.0%	5

15 Timer

```
class Timer

instance variables
 time : nat := 0;
 private static timerInstance : Timer := new Timer();
```

```
values
 stepLength : nat = 1;

operations


public static getInstance: () ==> Timer
getInstance() ==
  return timerInstance;


public stepTime : () ==> ()
stepTime() ==
  time := time + stepLength;


public getTime : () ==> nat
getTime() ==
  return time;

end Timer
```

Function or operation	Line	Coverage	Calls
getInstance	12	100.0%	3
getTime	20	100.0%	45
stepTime	16	0.0%	0
Timer.vdmpp		100.0%	48

16 World

```
class World

instance variables

public static env: [Environment] := nil;
public static timeRef : Timer := Timer`getInstance();

operations

public World : () ==> World

World() ==
 env := new Environment("scenario.txt");

public Run : () ==> ()

Run() ==
 (env.Run(););

end World
```

Function or operation	Line	Coverage	Calls

Run	15	100.0%	2
World	11	100.0%	2
World.vdmpp		100.0%	4