Design of Information Systems

Management System for Vehicle-Operation Training Organizations

Universität Bremen

University of Bremen
Computer Science
Database Systems Group

Summer 2018

Examined by
Prof. Dr. Martin Gogolla
Andreas Kästner
Contents

List of Figures VII

1 Introduction 1
   1.1 Purpose of this Report ............................... 1
   1.2 Domain and Developed System ......................... 1

2 Class Structure 2
   2.1 Object to Class ..................................... 2
   2.2 Final Class Structure ................................ 2
      2.2.1 Date ........................................... 8
      2.2.2 Person ........................................ 8
      2.2.3 Address ....................................... 9
      2.2.4 License ........................................ 9
      2.2.5 Qualification .................................. 9
      2.2.6 Programme ..................................... 10
      2.2.7 Unit ......................................... 10
      2.2.8 TheoryUnit ................................... 10
      2.2.9 PracticalUnit .................................. 10
      2.2.10 Reservation .................................. 11
      2.2.11 Vehicle ....................................... 11
      2.2.12 Class ......................................... 12
      2.2.13 Charter ....................................... 12
      2.2.14 AircraftCharter ................................ 13
      2.2.15 Attendance ................................... 13

3 Invariants 13
   3.1 Date ................................................ 14
   3.2 Person .............................................. 14
      3.2.1 Positive Person ID .............................. 14
      3.2.2 Unique Person ID ............................... 14
   3.3 License ............................................. 14
      3.3.1 Unique License ................................ 14
   3.4 Programme .......................................... 14
      3.4.1 Unique Programme ................................ 14
      3.4.2 TheoryUnit ..................................... 14
      3.4.3 Student Not Instructor .......................... 14
   3.5 PracticalUnit ....................................... 15
      3.5.1 Student Not Instructor .......................... 15
3.6 Reservation ................................................. 15
   3.6.1 Student Not Instructor ................................. 15
   3.6.2 Valid Beginning and Ending .......................... 15
   3.6.3 No Chronological Overlapping ........................ 15
3.7 Vehicle .................................................... 15
   3.7.1 Unique Vehicle ......................................... 15
   3.7.2 Non-negative Price ..................................... 15
3.8 Class ....................................................... 16
   3.8.1 Unique Class ........................................... 16
3.9 Charter ..................................................... 16
   3.9.1 No Chronological Overlapping ........................ 16
   3.9.2 Valid Departure and Arrival .......................... 16
3.10 AircraftCharter ........................................... 16
   3.10.1 Positive Number of Landings ........................ 16
   3.10.2 Valid Take-off and Landing ........................... 16
3.11 Further Ideas ............................................. 16
4 Operations .................................................... 17
  4.1 Date ....................................................... 17
     4.1.1 equals ................................................. 17
     4.1.2 before .................................................. 17
     4.1.3 after ................................................. 18
     4.1.4 duration .............................................. 19
  4.2 Person ...................................................... 19
     4.2.1 init ................................................... 19
     4.2.2 isQualifiedForProgramme ............................ 21
     4.2.3 isQualifiedForVehicle ................................ 21
     4.2.4 addCharter ............................................ 21
     4.2.5 addSupervisedCharter ................................. 22
     4.2.6 sumOfHours ........................................... 22
  4.3 Programme ................................................ 23
     4.3.1 qualifiedInstructors ................................. 23
  4.4 Charter .................................................... 23
     4.4.1 duration .............................................. 23
5 Test Cases .................................................... 23
  5.1 Positive Scenarios ....................................... 25
     5.1.1 Positive 1 ............................................. 25
     5.1.2 Positive 2 ............................................. 27
  5.2 Negative Scenarios ....................................... 29
# List of Figures

1. UML Object Diagram modeling parts of an aviation academy  ...  3
2. UML class diagram generated from the object diagram displayed in figure 1 ...  4
3. UML object diagram modeling parts of a driving school  ...  5
4. UML class diagram generated from the object diagram displayed in figure 3 ...  6
5. Final UML class diagram  ...  7
6. Initial state  ...  24
7. Object and link coverage  ...  25
8. Positive 1  ...  26
9. Invariants in Positive 1  ...  27
10. Positive 2  ...  28
11. Invariants in Positive 2  ...  29
1. Introduction

1.1. Purpose of this Report

This report documents a case study written as a practical part of Professor Martin Gogolla's lecture “Design of Information Systems” in summer 2018. The document is written according to the tasks outlined on the lecture’s website (Gogolla and Kästner 2018).

1.2. Domain and Developed System

The vast majority of driving schools relies on software programs for scheduling lessons and their student administration. According to this high demand many highly professional standard software solutions are offered for organizations in driver’s education. In contrast, for related domains like flight schools and boating schools, there are not as many tools available.

As the market for information systems for aviation academies, for example, is rather small, so is the number of suitable tools. Often the offered programs started as ‘hobby projects’ originating from aviation clubs that expanded to the developers’ side jobs over the years.

However, most requirements of all training organizations like driving schools, flight schools or boating schools are the same. On the one hand, the data of students aiming for certain licences have to be administrated. Practical lessons on vehicles (maybe aircraft or watercraft) and their maintenance have to be coordinated, on the other hand. In addition, especially flight and boating schools offer to charter their craft for former students or other trained customers in order to maximize the vehicles’ operating grades.

As in most cases, it does not matter whether a training is for cars, aeroplanes or sailing yachts, this project aims for a system that—prescinding from the individual vehicles’ details—unifies the requirements of all those organizations performing training related to any vehicle operation. In fact, it should be possible to model the individual business based on a common data model. As a result, the economically less interesting domains could benefit from the popular ones and also be provided with a professional software system.
2. Class Structure

As this project aims for a generic model covering the requirements of any kind of training organization related to vehicle operation training, it is assumed the modelled organization performs training on multiple vehicle types. This might not represent the reality but ensures a universal model. Actually, there are driving schools that offer training for road vehicles and boats and at least one German company offers training for all common road vehicles, for boats and even for aircraft, which is very unusual (Fahrschule Norbert Klippel 2011).

2.1. Object to Class

For a start, object models of two concrete scenarios were developed. The first object diagram (figure 1) displays parts of an aviation academy. The second object diagram (figure 3) shows parts of a driving school, focussing on the training programme and its units.

Applying the USE ObjectToClass plugin’s function Transform to class diagram these object diagrams were automatically transformed to preliminary class diagrams (figures 2 and 4) which finally helped to design the actual class diagram for the developed system (see section 2.2).

Generalizing examples is a useful method for approaching a universal model. Unfortunately, due to the USE tool’s usability, the process took much longer than it should have. In the “Object to Class” mode the diagram’s layout is not saved automatically. It has to be saved manually on every change and loaded manually when continuing the design of the model. Now and then USE even crashes, just printing exception stack traces to the console. Presumably, manual modelling of simple object diagrams and a manual transformation would have been much faster.

In contrast, the design of UML class diagrams in USE (like designed for section 2.2) is very comfortable. Editing the textual .use file is much more efficient than the interaction with many other UML-based CASE tools.

2.2. Final Class Structure

The class diagrams generated from the object diagrams led to the class structure displayed by the UML diagram in figure 5.

Below the contained classes with their attributes and associations are outlined.
Figure 1: UML Object Diagram modeling parts of an aviation academy
Figure 2: UML class diagram generated from the object diagram displayed in figure 1
Figure 3: UML object diagram modeling parts of a driving school
Figure 4: UML class diagram generated from the object diagram displayed in figure 3
Figure 5: Final UML class diagram
2.2.1. Date

As the system deals a lot with booking, many features are based on date and time information. A Date object stores a point in time.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Integer</td>
<td>The year.</td>
</tr>
<tr>
<td>month</td>
<td>Integer</td>
<td>The month.</td>
</tr>
<tr>
<td>day</td>
<td>Integer</td>
<td>The day.</td>
</tr>
<tr>
<td>hour</td>
<td>Integer</td>
<td>The hour.</td>
</tr>
<tr>
<td>minute</td>
<td>Integer</td>
<td>The minute.</td>
</tr>
</tbody>
</table>

Alternatives:

Date and time could also be encoded in either string or integer attributes in the objects where a date information is required. By default, there are not any compare operations on strings apart from a check on equality and any custom implementation of compares would be more than complicated. In contrast, integer values with a suitable encoding like "yyyymmddHHss" could be compared by using standard compare operations (>, <, >=, etc.). Unfortunately, such a number does not fit into USE's Integer type, which is why Date was created.

2.2.2. Person

A Person is any person related to the modelled organization. There is no difference between students and instructors. These characteristics are modelled by associations (an instructor may need a certain license himself in order to act as an instructor, for example).

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateOfBirth</td>
<td>Integer</td>
<td>The person's date of birth.</td>
</tr>
<tr>
<td>eMailAddress</td>
<td>String</td>
<td>The person’s email address.</td>
</tr>
<tr>
<td>firstName</td>
<td>String</td>
<td>The person’s first name.</td>
</tr>
<tr>
<td>lastName</td>
<td>String</td>
<td>The person’s last name.</td>
</tr>
<tr>
<td>gender</td>
<td>String</td>
<td>The person’s gender.</td>
</tr>
<tr>
<td>id</td>
<td>Integer</td>
<td>The person’s identification number (personnel / customer number).</td>
</tr>
</tbody>
</table>
Associations:

- A person has an Address.
- A person may hold licenses (see Qualification).
- A person may follow a (training) Programme.
- A person may have placed bookings (see Reservation).
- A person may be expected to supervise Reservations as an instructor.
- A person may have supervised Reservations as an instructor.
- A person may actually have received units (see Charter).
- A person may have attended theory units (see Attendance).
- A person may have taught theory units (see TheoryUnit).

2.2.3. Address

Address is a data class storing the elements of a mailing address. Every Person refers to an own Address object even if multiple persons happen to have the same address.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>street</td>
<td>String</td>
<td>The street and house number.</td>
</tr>
<tr>
<td>zipCode</td>
<td>String</td>
<td>The ZIP code (&quot;postcode&quot;)</td>
</tr>
<tr>
<td>city</td>
<td>String</td>
<td>The city.</td>
</tr>
<tr>
<td>country</td>
<td>String</td>
<td>The country.</td>
</tr>
</tbody>
</table>

2.2.4. License

A License authorizes a Person to operate a craft or to give lessons for a specific craft.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>String</td>
<td>The license's title.</td>
</tr>
</tbody>
</table>

2.2.5. Qualification

As an association class, Qualification links Persons with Licenses.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expiryDate</td>
<td>Date</td>
<td>The day the qualification expires.</td>
</tr>
</tbody>
</table>
2.2.6. Programme

A Programme combines all lessons that have to be learned in order to get a specific License.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>String</td>
<td>The programme’s title.</td>
</tr>
</tbody>
</table>

Associations:

- A programme aims for a License.
- An instructor teaching a programme requires a certain License.
- A programme (curriculum) consists of Units.

2.2.7. Unit

This class is meant to be abstract. Unfortunately, USE does not export it’s abstract status to the PDF.\(^1\) A Unit can either be a TheoryUnit or a PracticalUnit.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>String</td>
<td>The unit’s title.</td>
</tr>
<tr>
<td>time</td>
<td>Integer</td>
<td>The time in minutes planned for this unit.</td>
</tr>
</tbody>
</table>

2.2.8. TheoryUnit

A TheoryUnit teaches theoretical subjects.

2.2.9. PracticalUnit

A PracticalUnit is a lesson taught on a vehicle.

\(^1\)Apparently only on Mac systems.
2.2.10. Reservation

A Reservation stores a Person’s reservation for a Vehicle.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginning</td>
<td>Date</td>
<td>Start of the booked period.</td>
</tr>
<tr>
<td>ending</td>
<td>Date</td>
<td>End of the booked period.</td>
</tr>
<tr>
<td>maintenance</td>
<td>Boolean</td>
<td>Whether the booking is for maintenance (e.g. workshop or cleaning) and therefore has not been booked by a student or customer.</td>
</tr>
</tbody>
</table>

Associations:

- A Reservation always refers to a Vehicle.

Alternatives:

To model a reservation that actually is a planned maintenance—a time the vehicle is in a workshop—it would also be possible to derive another class from Reservation.

2.2.11. Vehicle

A Vehicle is any kind of land craft, watercraft or aircraft that can be chartered.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>String</td>
<td>The vehicle’s id (most likely taken from number plate for cars or call sign for watercraft and aircraft).</td>
</tr>
<tr>
<td>pricePerHour</td>
<td>Real</td>
<td>The vehicle’s hourly price.</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>The vehicle type (e.g. brand and model).</td>
</tr>
<tr>
<td>clear</td>
<td>Boolean</td>
<td>Whether the vehicle can be used or is defective.</td>
</tr>
</tbody>
</table>

Associations:

- A Vehicle is of a Class.
- To operate a Vehicle, Licenses might be required.
2.2.12. Class

Each vehicle belongs to a specific Class. Some examples are listed below:

- Land craft:
  - Motorcycle
  - Car
  - Heavy Goods Vehicle
  - Bus
  - etc.
- Watercraft:
  - Motorboat
  - Dinghy
  - Sailing Yacht
  - etc.
- Aircraft:
  - Single Engine Piston
  - Touring Motor Glider
  - etc.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the class.</td>
</tr>
</tbody>
</table>

2.2.13. Charter

Charter stores the data that is needed to determine a price for a lesson. For domains where not complete lessons but minutes are charged—like in aviation—being able to store start and end is very important. A driving school probably would enter standard times from their timetable regardless of the fact that not all lessons exactly took 45 minutes.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>departure</td>
<td>Date</td>
<td>Time of departure.</td>
</tr>
<tr>
<td>arrival</td>
<td>Date</td>
<td>Time of arrival.</td>
</tr>
<tr>
<td>maintenance</td>
<td>Boolean</td>
<td>Whether the vehicle has been moved for maintenance or not (the driver or pilot might not have to pay for the “charter” when it was operationally necessary).</td>
</tr>
</tbody>
</table>
• A Charter always refers to one Vehicle.
• During a Charter exercises of Units might have been trained.

2.2.14. AircraftCharter

As there are some more attributes required to describe aircraft charter, a class AircraftCharter has been derived from Charter.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>takeOff</td>
<td>Date</td>
<td>The take-off time. In contrast, the attribute departure in the superclass denotes the time when the aircraft began to taxi.</td>
</tr>
<tr>
<td>landing</td>
<td>Date</td>
<td>The landing time. In contrast, the attribute arrival in the superclass denotes the time when the aircraft reached its parking position.</td>
</tr>
<tr>
<td>origin</td>
<td>String</td>
<td>The flight’s origin.</td>
</tr>
<tr>
<td>destination</td>
<td>String</td>
<td>The flight’s destination.</td>
</tr>
<tr>
<td>numberOfLanding</td>
<td>Integer</td>
<td>The number of landings performed during the flight, which might be more than the minimum of one when touch-and-go manoeuvres have been trained.</td>
</tr>
</tbody>
</table>

Associations: Inherited from superclass Charter.

Alternatives:
The aircraft specific attributes could belong to the standard class Charter. In that case they would be obsolete in cases not involving aircraft.

2.2.15. Attendance

Objects of this association class store when a student (Person) attended a TheoryUnit.

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>Date</td>
<td>Start of the lesson.</td>
</tr>
<tr>
<td>end</td>
<td>Date</td>
<td>End of the lesson.</td>
</tr>
</tbody>
</table>

3. Invariants
3.1. Date

There are no invariants checking the correct use of Date as a full calendar model would be required for sensible constraints.

3.2. Person

3.2.1. Positive Person ID

A person’s ID must be greater than or equal to zero.

\[
\text{inv positivePersonId:} \quad \text{self.id} \geq 0
\]

3.2.2. Unique Person ID

A person’s ID must be unique.

\[
\text{inv uniquePersonId:} \quad \text{Person.allInstances} \rightarrow \text{isUnique(id)}
\]

3.3. License

3.3.1. Unique License

A license must be unique.

\[
\text{inv uniqueLicense:} \quad \text{License.allInstances} \rightarrow \text{isUnique(title)}
\]

3.4. Programme

3.4.1. Unique Programme

A training programme must be unique.

\[
\text{inv uniqueProgramme:} \quad \text{Programme.allInstances} \rightarrow \text{isUnique(title)}
\]

3.4.2. TheoryUnit

3.4.3. Student Not Instructor

A student must not be his or her own instructor.

\[
\text{inv studentNotInstructorInTheoryUnit:} \quad \text{not self.theoryUnit.instructor} \rightarrow \text{asSet()} \rightarrow \text{includes(self)}
\]
3.5. PracticalUnit

3.5.1. Student Not Instructor

A student must not be his or her own instructor.

\[ \text{inv studentNotInstructorInPracticalUnit:} \]
\[ \text{not self.unit.unitType.instructor} \rightarrow \text{asSet()} \rightarrow \text{includes(self)} \]

3.6. Reservation

3.6.1. Student Not Instructor

A student must not be his or her own instructor.

\[ \text{inv studentNotInstructorInReservation:} \]
\[ \text{not self.booking.instructor} \rightarrow \text{asSet()} \rightarrow \text{includes(self)} \]

3.6.2. Valid Beginning and Ending

\[ \text{inv validBeginningAndEnding:} \]
\[ \text{self.beginning} \leq \text{self.ending} \]

3.6.3. No Chronological Overlapping

\[ \text{inv noChronologicalOverlappingOfReservations:} \]
\[ \text{self.craft.booking} \rightarrow \text{excluding(self)} \rightarrow \text{forAll(} \]
\[ \text{r|r.ending < self.beginning} \]
\[ \text{or r.beginning > self.ending} \]
\[ \text{)} \]

3.7. Vehicle

3.7.1. Unique Vehicle

A vehicle ID must be unique.

\[ \text{inv uniqueVehicle:} \]
\[ \text{Vehicle.allInstances} \rightarrow \text{isUnique(id)} \]

3.7.2. Non-negative Price

A vehicle's price must not be negative.

\[ \text{inv nonNegativePrice:} \]
\[ \text{self.pricePerHour} \geq 0 \]
3.8. Class

3.8.1. Unique Class

A (vehicle) class must be unique.

inv uniqueClass:
  Class.allInstances -> isUnique(name)

3.9. Charter

3.9.1. No Chronological Overlapping

inv noChronologicalOverlappingOfCharters:
  self.craft.unit -> excluding(self) -> forAll(
    c | c.arrival < self.departure
    or c.departure > self.arrival
  )

3.9.2. Valid Departure and Arrival

inv validDepartureAndArrival:
  self.departure <= self.arrival

3.10. AircraftCharter

3.10.1. Positive Number of Landings

The number of landings during an aircraft charter must be greater than zero.

inv positiveLandings:
  self.numberOfLandings > 0

3.10.2. Valid Take-off and Landing

inv validTakeOffAndLanding:
  self.takeOff <= self.landing
  and self.takeOff >= self.departure
  and self.landing <= self.arrival

3.11. Further Ideas

- Checking whether a Person either holds the required license for a booked Vehicle or planned a supervised unit with an instructor.
- An instructor must have the required license to teach a programme’s units. This constraint may be problematic as an instructor might somehow lose his or her license (due to age or medical issues, for example). Then instantly all
lessons taught in the past become invalid. This is why an operation `isQualifiedForProgramme` has been defined. It can be used as a precondition for the operation entering a unit.

4. Operations

4.1. Date

4.1.1. equals

The operation checks whether a `Date` equals another `Date`.

Parameters:
- `d : Date` – The other date.

Returns:
Boolean: `true`, if the instance equals `d`; `false` else.

Code:
```
equals(d : Date) : Boolean
= self.year = d.year and
  self.month = d.month and
  self.day = d.day and
  self.hour = d.hour and
  self.minute = d.minute
```

4.1.2. before

The operation checks whether a `Date` represents a date prior to another `Date`.

Parameters:
- `d : Date` – The other date.

Returns:
Boolean: `true`, if the instance represents a date prior to `d`; `false` else.

Code:
```
before(d : Date) : Boolean
= if not (self.year = d.year) then
  self.year < d.year
else
  
```
if not (self.month = d.month) then
    self.month < d.month
else
    if not (self.day = d.day) then
        self.day < d.day
    else
        if not (self.hour = d.hour) then
            self.hour < d.hour
        else
            self.minute < d.minute
        endif
    endif
endif
endif
endif

Preconditions

pre initialized:
    not self.year.isUndefined() and
    not self.month.isUndefined() and
    not self.day.isUndefined() and
    not self.hour.isUndefined() and
    not self.minute.isUndefined()

4.1.3. after

The operation checks whether a Date represents a date later than another Date.

Parameters:

- d : Date – The other date.

Returns:

Boolean: true, if the instance represents a date later than d; false else.

Code:

after(d : Date): Boolean = not self.before(d)

Preconditions

pre initialized:
    not self.year.isUndefined() and
    not self.month.isUndefined() and
    not self.day.isUndefined() and
not self.hour.isDefined() and
not self.minute.isDefined()

4.1.4. duration
The operation calculates the period’s time between a Date and another Date. The operation only works for dates of the same day.

Parameters:

- d : Date – The other date.

Returns:
Integer: The time between the two dates in minutes.

Code:

duration(d : Date) : Integer
    = (self.hour - d.hour).abs() * 60
    + (self.minute - d.minute).abs()

Preconditions
pre initialized:
    not self.year.isDefined() and
    not self.month.isDefined() and
    not self.day.isDefined() and
    not self.hour.isDefined() and
    not self.minute.isDefined()

4.2. Person
4.2.1. init
Initializes a Person object (“constructor”).

Parameters:

- id : Integer – The person’s id.
- lastName : String – The person’s last name.
- firstName : String – The person’s first name.
• dateOfBirth : Date – The person’s date of birth.

• eMailAddress : String – The person’s email address.

• residence : Address – The person’s (post) address.

Code:

```javascript
init (id : Integer,
      lastName : String,
      firstName : String,
      gender : String,
      dateOfBirth : Date,
      eMailAddress : String,
      residence : Address)
begin
  self.id := id;
  self.lastName := lastName;
  self.firstName := firstName;
  self.dateOfBirth := dateOfBirth;
  self.eMailAddress := eMailAddress;
  insert (self, residence) into livesAt
end
```

Preconditions:

pre notInitialized:
  self.id.isUndefined() and
  self.lastName.isUndefined() and
  self.firstName.isUndefined() and
  self.gender.isUndefined() and
  self.dateOfBirth.isUndefined() and
  self.residence.isUndefined()

Postconditions:

post initialized:
  self.id = id and
  self.lastName = lastName and
  self.firstName = firstName and
  self.dateOfBirth = dateOfBirth and
  self.eMailAddress = eMailAddress and
  self.residence = residence
4.2.2. isQualifiedForProgramme

Checks whether a Person is qualified for teaching a specified Programme.

Parameters:

- prog : Programme – The programme that shall be checked.

Returns:

Boolean: true if the person is qualified for teaching the programme; false else.

Code:

```latex
isQualifiedForProgramme(prog : Programme) : Boolean
= let heldLicenses = self.heldLicense in
  heldLicenses -> union(prog.instructorLicense -> asSet())
= heldLicenses
```

4.2.3. isQualifiedForVehicle

Checks whether a Person is qualified for operating specified Vehicle, i.e. if the person holds the required License(s).

Parameters:

- vhcl : Vehicle – The vehicle that shall be checked.

Returns:

Boolean: true if the person is qualified for operating the vehicle; false else.

Code:

```latex
isQualifiedForVehicle(vhcl : Vehicle) : Boolean
= let heldLicenses = self.heldLicense in
  heldLicenses -> union(vhcl.qualification -> asSet())
= heldLicenses
```

4.2.4. addCharter

Adds a Charter.

Parameters:

- charter : Charter – The charter.
4.2.5. **addSupervisedCharter**

Adds a *Charter* that has been supervised by an instructor.

**Parameters:**

- charter : Charter – The charter.
- instructor : Person – The instructor.
- unit : PracticalUnit – The unit the performed exercises were taken from.

**Code:**

```java
addSupervisedCharter(charter : Charter,
                      instructor : Person,
                      unit : PracticalUnit)
begin
    self.addCharter(charter);
    insert (charter, unit) into was;
    insert (instructor, charter) into supervised;
end
```

**Preconditions:**

```java
pre validInstructor:
    instructor.heldLicense->union(unit.curriculum.
                                  instructorLicense)->asSet() = instructor.heldLicense
```

4.2.6. **sumOfHours**

Sums up the total duration of all units a *Person* has received.

22
‘Hours’, in the operation’s title, should not be taken literally as the function —with the Date’s class present implementation—returns minutes.

Returns:
Integer: Total duration in minutes.

Code:
```
sumOfHours() : Integer = self.unit.duration() -> sum()
```

4.3. Programme

4.3.1. qualifiedInstructors

Returns all Persons who are qualified to teach lessons in the Programme.

Returns:
Set(Person): All persons who are qualified to teach lessons in the programme.

Code:
```
qualifiedInstructors() : Set(Person) =
    Person.allInstances() -> select(p : Person |
        p.heldLicense -> intersection(self.instructorLicense)
        = self.instructorLicense
    )
```

4.4. Charter

4.4.1. duration

Returns a Charter’s duration.

Returns:
Integer: The charter’s duration.

Code:
```
duration() : Integer = arrival.duration(departure)
```

5. Test Cases

Two positive scenario and one negative scenario have been designed in order to test the system. All scenarios require an initial state (see section A.2.1). Figure 6 shows an object diagram of the initial state.
Figure 6: Initial state
5.1. Positive Scenarios

In the positive scenarios, all invariants, pre- and postconditions and inherent constraints are fulfilled.

The positive scenarios together cover all objects and all links as shown in figure 7. Only Unit is never instantiated because it is an abstract class.

![Object and link coverage](image)

5.1.1. Positive 1

The scenario is given by the listing in section A.2.2. Figure 8 shows the scenario as UML object diagram. As shown in figure 9, all invariants are fulfilled.
Figure 8: Positive 1
5.1.2. Positive 2

The scenario is given by the listing in section A.2.3. Figure 10 shows the scenario as UML object diagram. As shown in figure 11, all invariants are fulfilled.
Figure 10: Positive 2
5.2. Negative Scenarios

In negative scenarios at least one condition (invariant, pre- or postcondition) fails.

5.2.1. Negative 1

This scenario tries to add a supervised charter with the student himself as an instructor. As expected, this is not possible:

[Error] 1 precondition in operation call ‘Person::
    addSupervisedCharter(self:student, charter:lesson,
    instructor:student, unit:emergencyLanding)’ does not hold:
validInstructor: ...

6. Queries

Queries—in contrast to operations—do not deal with individual objects. They rather provide information on the entire record.

6.1. Qualified Instructors

This query returns all persons in the organization who are qualified to instruct in any training programme.

 Requires:
init.soil

 Query:
Programme.allInstances().qualifiedInstructors()->asSet()

Result:
Set{instructor} : Set(Person)
References
