

8.4 A travel booking system

In the following we show another example using subclasses and virtual methods by making a model of elements of a travel agency that helps people making bookings.

There are many kinds of bookings; however in this example we cover a subset: travel bookings like flight bookings, and train bookings, bookings of hotel, and bookings of roundtrips. Some of them can be done via the internet, while others, like roundtrips, may require help from a travel agent.

The experience from the account example tells us that all these different types of booking have properties in common. This is modeled by a class `Booking` with all the common properties, like booking reference, reservation date, customer, handling agent and the last day of cancellation. The different types of bookings are then defined as subclasses of `Booking`.

In the examples below, we use a number of auxiliary classes that are not specified, and as usual, this is indicated by ...

```
class Booking(bookingRef: var String):  
    reservationDate: var Date  
    customer: ref Person  
    agent: ref Employee  
    lastCancelDate: var Date
```

```
class Person: ...  
class Employee: ...
```

`bookingRef` is a string that uniquely identifies the booking. It is issued by e.g. the airline company and typically has to be used in order to cancel a booking.

Note that `reservationDate` represents the date at which the booking is made at the travel agency, not e.g. the departure date for a travel booking.

For all bookings there are three activities: register the booking in a booking register of the travel agency, confirm the booking, and cancel the booking.

Assuming that the travel agency keeps a register of bookings:

```
bookingRegister: obj Set(Booking)
```

the following action of `Booking` inserts the booking in this register:

```
class Booking:  
    _"  
    bookingRegister.insert(this(Booking))
```

Statements in a class description, as the method call above, are executed when a `Booking` object is generated.

Cancelling a booking is not the same for all types of bookings. This is reflected by defining the method `cancel` as a virtual method:

```

class Booking:
    "-"
    cancel:<
        if (today > lastCancelDate) :then
            console.print("cancellation not possible")
        :else
            bookingRegister.remove(this(Booking))
            inner(cancel)

```

However, there are some common things to be done for every type of cancellation, and that is specified in `cancel`, e.g. that cancellation is not possible after the last cancellation date, and that cancellation amounts to remove the booking from the `bookingRegister`.

The special things to be done for the different types of bookings is represented by `inner(cancel)`, e.g. that the carriers or the hotel has to be informed that the booking has been cancelled.

The method `confirm` also depends on which type of booking to confirm, so that has to be defined specifically for each subclass of `Booking`. This is indicated by defining `confirm` as a virtual method. Even though it has to be defined specifically for each subclass of `Booking`, there are some common actions of `confirm`, and these are described as part of the virtual method:

```

class Booking:
    "-"
    confirm -> confirmText: var String:<
        confirmText:= "Booking " + bookingRef + ":" '\n'
        inner(confirm)
        confirmText := confirmText + "has been confirmed" + '\n'

```

The `confirm` method computes a text that consist of the details of the `Booking`. This text is returned by `confirm` in the variable `confirmText`.

Common to all bookings is that the confirmation text has the same text around the details of the actual booking. Execution of `inner` implies the execution the statements of the `confirm` of the given subclass, and this will add the detailed text of the `confirmText`. This implies that the special `confirm` methods in different subclasses are extensions of the actions of the virtual method `confirm`.

The travel agent may decide how to use the confirmation text: to print it for off-line checking, include it in a travel document, or send it by email to the customer.

A typical scenario in this example is that the travel agency is asked to find a travel from a city (origin) to another city (destination), be it by air or by train.

A `TravelBooking` object therefore has an `origin` and `destination` city:

```

class TravelBooking: Booking
    origin, destination: ref City
    departureDate: var Date
    confirm:<
        confirmText := confirmText +
            "from " + origin.name + "to " + destination.name '\n'
        "at " + departureDate
        inner(confirm)

```

We do not go into details about City, we simply assume that it is represented by an instance of class City:

```
class City:
  name: var String
  ...
```

The agency handles two types of travel bookings, and these are represented by two subclasses of TravelBooking. In addition to origin and destination, these include the fact that cities may have more than one airport and often have more than one train station. At this point do not include details about flight or train routes with row and seat of the booking. We will later introduce a model of flight routes and flights.

```
class FlightBooking: TravelBooking
  fromAirport, toAirport: ref Airport
  theCarrier: ref Airline
  cancel::
    theCarrier.notifyCancellation(bookingRef)
  confirm::
    confirmText := confirmText +
      "from " + fromAirport.name + "to " + toAirport.name '\n'
```

```
class TrainBooking: TravelBooking
  fromStation, toStation: ref TrainStation
  theCarrier: ref TrainCompany
  cancel::
    theCarrier.notifyCancellation(bookingRef)
  confirm::
    confirmText := confirmText +
      "from " + fromStation.name + "to " + toStation.name '\n'
```

As above, this assumes the following classes:

```
class Airport:
  name: var String
  ...
class TrainStation:
  name: var String
  ...
```

None of these classes, except Flight, are described in this book. Class Flight is defined in section .

Objects of class HotelBooking represent bookings of stays at a hotel. They have properties like thehotel: a reference to an object representing the hotel, room: the room number, arrivalDate, and noOfNights:

```
class HotelBooking: Booking
  theHotel: ref Hotel
  arrivalDate: var Date
  room: var Integer
  noOfNights: var Integer
```

```

confirm::
  confirmText := confirmText +
    "at " + theHotel.name '\n' +
    "from " + arrivalDate + noOfNights " nights" '\n'

```

This assumes a class `Hotel` with at least a name attribute:

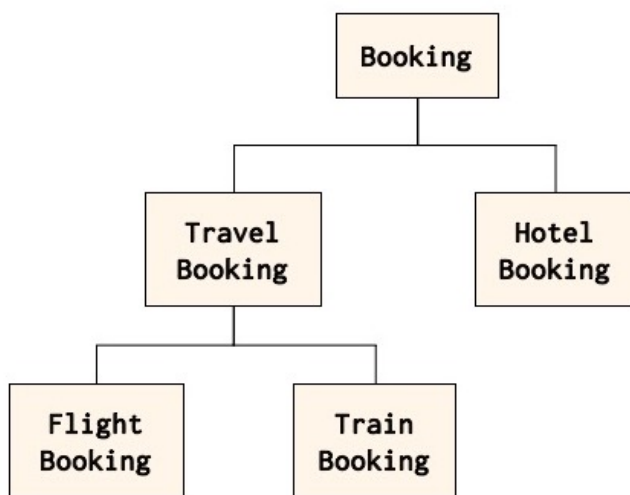
```

class Hotel:
  name: var String
  addr: obj Addresss
  ...

```

Class `Address` is not specified here, but a version of it is defined in section .

The following diagram illustrates the hierarchy of `Booking` classes:



Booking Class Hierarchy

The following is a sketch of the sequence of actions that has to be performed in order to make a `FlightBooking`, triggered somehow by an agent, given actual values of the following items:

```

theCustomer: ref Person
origin: ref City,
destination: ref City,
from, to: ref Airport,
departureDate: var Date

```

Finding a carrier that has flights from origin to destination cities is done by invoking a method `findAirlineCarrier`:

```

airlineCarrier: ref Airline

```

```

airlineCarrier := findAirlineCarrier(origin, destination)

```

We do not provide the details of `findAirlineCarrier`

```

findAirlineCarrier(origin, destination: ref City) -> carrier: ref Airline:
  ...

```

Next a flight is booked by invoking `bookFlight` of `airlineCarrier`, providing a booking reference:

```
bookingRef: var String
```

```
bookingRef := airlineCarrier.bookFlight(from, to, departureDate)
```

The method `bookFlight` has this signature:

```
bookFlight(from, to: ref Airport, departureDate: var Date)-> bookingRef: var String:  
    ...
```

Given the value of `bookingRef` it is now possible to generate a `FlightBooking` object:

```
booking: ref FlightBooking
```

```
booking := FlightBooking(bookingRef)  
booking.customer := theCustomer  
booking.carrier := theCarrier  
booking.origin := origin  
...  
booking.registerBooking  
booking.confirm  
theCustomer.sendEmail(booking.confirmText)
```

To complete the example, we need to represent waggon and seat number for train reservation and row and seat number for flight reservations. We return to this in section .

Bookings of round trips is where the travel agents really come into play. A round trip is not a special type of booking, but rather consist of a list of bookings. A roundtrip is based on an itinerary that either is presented by the customer as the desired/planned journey, or is worked out in a cooperation between the customer and a travel agent. As a round trip in this example includes both travel bookings and hotel bookings, a round trip is represented by an object of the class `RoundTrip`. The trip is an ordered list of bookings, i.e. not just a set, as there is a sequence of travels and hotel stays.

```
class RoundTripBooking:  
  plan: ref Itinerary  
  trip: obj OrderedList(Booking)  
  confirm:  
  trip.scan  
  b.confirm  
class Itinenary:  
  ...
```