Assignment 4: Object creation and logic

ETH Zurich

Hand-out: Monday, 8 October 2012
Due: Wednesday, 17 October 2012

Spiked Math © Mike (http://spikedmath.com/445.html)

Goals

• Create new objects.
• Use boolean operators to express interesting conditions.
• Write contracts.

1 Creating objects in Traffic

Up until now you were either working with predefined objects, such as Zurich or Polybahn, or you asked a CITY object to create an object for you, as in Zurich.add_station ("Zoo", 1800, 500). In this assignment, you will create new objects yourself and add them to Zurich. To achieve that, use the following general scheme:

1. Declare an attribute [Touch Of Class, page 238] or a local variable [Touch Of Class, page 233] of the required type. For example:

```plaintext
class OBJECT_CREATION

feature -- Explore Zurich
    building: BUILDING
        -- An example of an attribute.
```
explore
   -- Create new objects for Zurich.
local
   p: VECTOR  -- An example of a local variable.
do
   ...
   end
end

Prefer local variables over attributes unless the object is used by more than one feature or you need to retain the object value between feature executions.

2. Create the object using one of the creation procedures [Touch Of Class, page 122] declared in the corresponding class. For example:

   class VECTOR

   create
      make, make_polar

   feature {NONE}  -- Initialization
      make (a_x, a_y: REAL_64)
         -- Create a vector with coordinates ‘a_x’ and ‘a_y’.
      ...
      end

   make_polar (a_length, a_angle: REAL_64)
      -- Create a vector with polar coordinates ‘a_length’ and ‘a_angle’.
      ...
      end

To create an object you can use an instruction such as:

   create p.make (250, -20)

You don’t have to create the object if its type is expanded. In this case the object will be automatically created for you. Examples of expanded types are INTEGER, BOOLEAN and REAL_64.

3. Add the object to the city by calling an appropriate command on the city, for example:

   Zurich.add_building (building)

Note that only real components of the city, such as stations, lines, buildings, routes and public transportation units, have to be added to the city. There is no need to add helper objects, which are only used to build other objects, such as vectors, colors, route segments and so on.

To do

1. Download http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/04/traffic.zip, unzip it and open assignment_4.ecf. Open class OBJECT_CREATION in the editor.

2. Declare a command add_buildings in OBJECT_CREATION, which adds a couple of buildings to Zurich, for example the ETH main building and the Opera house.

   Note that in order to create an object of class BUILDING, you first have to create two objects of class VECTOR that denote positions (with respect to the city center) of two opposite corners of a building, with respect to the city center.
3. Add a call to add_buildings from explore and check that the new buildings now appear on the map (see Figure 1 for an example).

![Figure 1: ETH main building on the map of Zurich](image)

4. Imagine that a friend of yours needs to get from the ETH main building to the Opera house, and also wants to pass by Confiserie Sprüngli at Paradeplatz on the way. You decided to help him and show him the public transportation directions on the map of Zurich. Declare a command add_route in OBJECT CREATION, which creates a route starting at station Polyterrasse, passing through Paradeplatz and ending at Opernhaus.

Note that in order to create an object of class ROUTE you first have to create one or more objects of class LEG: segments of the route that go along a particular line. For example, a route from Sihlpost to Bellevue can consist of two legs: the first going from Sihlpost to Central along tram line 3, and the second one going from Central to Bellevue along tram line 4. You can call queries station and line on Zurich to get the objects representing stations and lines that each leg goes through. To connect the legs to each other use either the link command of class LEG or the append_leg command of class ROUTE.

Don’t forget to add the new route to Zurich.

5. Add a call to add_route from explore and check that the route appears on the map.

To hand in
Hand in the code of the class OBJECT CREATION.

2 Temperature application

In this task you will write an application which converts temperatures between Celsius and Kelvin scales using the following formula:

\[ T_{\text{Celsius}} = T_{\text{Kelvin}} - 273 \]

The application should consist of two classes: TEMPERATURE and APPLICATION. Class TEMPERATURE encapsulates the notion of temperature; it hides from its clients implementation details such as how temperatures are converted between different scales. Class APPLICATION is a root class and a client of TEMPERATURE; it provides the user interface for the application.
To do

1. Launch EiffelStudio. Create a new project of type “Basic application (no graphics library included)”, using the settings shown in Figure 2 (feel free to pick any location).

![Figure 2: New project](image)

2. Download the skeleton classes for TEMPERATURE and APPLICATION from [http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/04/temperature.e](http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/04/temperature.e) and [http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/04/application.e](http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/04/application.e) and put them into your project directory. From the Project menu in EiffelStudio choose Find Added Classes & Recompile; this way EiffelStudio will see the new classes you added.

3. Fill in the missing pieces of classes TEMPERATURE and APPLICATION according to the comments. Feature execute of class APPLICATION must use the class TEMPERATURE to perform the conversion.

4. Express the following properties using contracts:
   - `make_celsius` does not accept values below -273 and sets the Celsius temperature to the provided value;
   - `make_kelvin` does not accept values below zero and sets the Kelvin temperature to the provided value;
   - a temperature can never be below zero on the Kelvin scale.
   - `average` does not accept the Void reference and the resulting average temperature lies somewhere in between the two initial temperatures.

An execution of your application should look similar to this:

```
Enter the first temperature in Celsius: 0
The first temperature in Kelvin is: 273
Enter the second temperature in Kelvin: 283
The second temperature in Celsius is: 10
The average in Celsius is: 5
The average in Kelvin is: 278
```
To hand in

Hand in the code of *TEMPERATURE* and *APPLICATION*.

3 Ein Ticket für alles

Imagine that you work for ZVV and you are writing an application that allows customers to find out if they are eligible for a discounted seasonal ticket. ZVV decided to make the tickets more affordable and introduced a really tricky rule: you can get a discount in one of three cases: 1) you are younger than 25 or 2) you live in the city of Zurich and work anywhere outside the city or 3) you work in the city of Zurich and live in the canton of Zurich, but outside the city.

To do

1. Download [http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/04/ticket.zip](http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/04/ticket.zip), unzip it and open `ticket.ecf`. Open class `APPLICATION` in the editor.

2. This program first asks the user for some data: his birth date and the postal codes of his home and work place. Then it analyzes the data and decides if the user can get a discount. Your task is to implement a couple of functions that help it make the decision.

3. Implement function `in_zurich_city`: it should say that a postal code belongs to the city of Zurich if its first two digits are 80. Use the function `in_zurich_canton` as an example.

4. Implement function `valid_postal_code`: it should say that a string represents a valid postal code if it’s not a Void reference, it consists of four characters and represents a natural number. **Hint:** use query `is_natural` of class `STRING`.

5. Implement function `age` that determines a user’s age based on his birth date. To achieve that, inside the function create another object of class `DATE` to represent today’s date (check the list of creation procedures of class `DATE` to find an appropriate one). **Hint:** use function `relative_duration` of `DATE` to get the difference between two dates.

6. Implement function `gets_discount` that decides if a customer with the currently entered parameters is eligible for discount, according to the tricky rule described above.

An execution of your application should look similar to this:

Enter birth date as dd/mm/yyyy: 20/05/1985
Enter home postal code: 8051
Enter work postal code: 8600

Eligible for discount: True

To hand in

Hand in the code of class `APPLICATION`. 
