Assignment 3: Of objects and features

ETH Zurich

Hand-out: 2 October 2009
Due: 13 October 2009

Calvin and Hobbes© Bill Watterson

Goals

- Understand the difference between a class and an object.
- Distinguish between queries and commands.
- Learn to read feature call instructions.
- Write more feature call instructions.

1 Classes vs. objects

To do

1.1 Try to describe the difference between a class and an object (1-2 sentences).
1.2 Find an analogy that captures the relationship between objects and classes in real life.

To hand in

Write down your answers (1.1 and 1.2) and hand them in.

2 Categorizing features

There are two main categories of features: queries and commands. In last week’s assignment you learned how to categorize features. In this week’s lecture you have a better look at their declaration in the class interface. The general patterns for queries and commands in a class interface are:
TYPE, TYPE₁, TYPEₙ are class names. In the case of an argument declaration, they will tell you the expected type of the arguments. In the case of a query, TYPE denotes the type of the object you get as an answer when calling the query. Note, once again, that the only way to distinguish between a query and a command is to look whether a feature returns an object (i.e. look for the return type in its declaration).

The examples given above are from Listing 1 that shows a shortened interface of class TRAFFIC_BUILDING. The argument declaration of set_size uses a short form for the declaration of its arguments. Instead of stating for each argument that it is of type REAL_64, it separates the identifiers by comma (instead of semicolon) and gives the type at the end. The short form can be used whenever there are two or more arguments of the same type appearing one after the other in the declaration. So the declaration set_size (a_width, a_height, a_depth: REAL_64) is equivalent to set_size (a_width: REAL_64; a_height: REAL_64; a_depth: REAL_64) and contains_point (a_x: REAL_64; a_y: REAL_64): BOOLEAN could also be written as contains_point (a_x, a_y: REAL_64): BOOLEAN.

Listing 1: Class TRAFFIC_BUILDING

defered class interface TRAFFIC_BUILDING

feature

    center: TRAFFIC_POINT
    -- Center of the building

    corner_1: TRAFFIC_POINT
    -- Lower left corner of the building

    ensure
        result_exists: Result /= Void

    contains_point (a_x: REAL_64; a_y: REAL_64): BOOLEAN
    -- Is point (‘a_x’, ‘a_y’) inside building?

    spotlight
        -- Highlight.
    ensure -- from TRAFFIC_CITY_ITEM
        highlighted: is_highlighted

    set_center (a_center: TRAFFIC_POINT)
Todo

In Listing 2 you find the class interface of TRAFFIC_TIME that is responsible for simulating time in the city, used for example for letting passengers move at a certain speed. Make two lists of features for this class interface: one for queries, the other for commands. Use the way described above to distinguish between queries and commands.

Listing 2: Class TRAFFIC_TIME

defered class interface TRAFFIC_TIME

feature  —— All features

pause

    —— Pause the time count.
    require
        is_time_running
    ensure
        not is_time_running

actual_time: TIME

    —— Simulated time

reset

    —— Reset the time to (0:0:0).
    ensure
        is_time_running = False
        actual_time.hour = 0
        actual_time.minute = 0
        actual_time.second = 0

duration (a_start_time, a_end_time: TIME): TIME_DURATION

    —— Duration from 'a_start_time' until 'a_time2'.
    —— Takes into account midnight.
    require
        both_exist: a_start_time /= Void and a_end_time /= Void
    ensure
        result_exists : Result /= Void
result_positive : Result.is_positive

speedup: INTEGER_32
--- Speedup to let the time run faster than the real time

set_speedup (a_speedup: INTEGER_32)
--- Set speedup to ‘a_speedup’.
require
a_speedup_valid: a_speedup >= 1
ensure
speedup_set: speedup = a_speedup

start
--- Start to count the time at (0:0:0).
require
not is_time_running
ensure
is_time_running

is_time_running: BOOLEAN
--- Is the time running?

resume
--- Resume the paused time.
require
not is_time_running
ensure
is_time_running

set (a_hour, a_minute, a_second: INTEGER_32)
--- Sets the time to (‘a_hour’; ‘a_minute’; ‘a_second’).
require
valid_time: a_hour >= 0 and a_minute >= 0 and a_second >= 0

invariant
actual_time.hour >= 0
actual_time.minute >= 0

end

3 Feature reading

In Task 2 you saw that feature declarations of queries always include the declaration of a return type. The return type is the type of the object that is returned as an answer when calling the query. This knowledge, in combination with the fundamental mechanism of program execution (applying a “feature” to an “object”), allows to build complex targets and arguments to feature call instructions. To make it clearer:

- Queries return a value (an object), e.g. Station_balard.location yields an object of type TRAFFIC_POINT, the position of Balard. Since the result is an object, it is possible to apply features to it, e.g. Station_balard.location.up_by (5.0). What features can be applied is defined in the class TRAFFIC_POINT. As a side note: Station_balard is
also a query returning an object of type `TRAFFIC_STATION` and is declared in class `TOUCH_PARIS_OBJECTS`. Class `PREVIEW` offers the features of `TOUCH_PARIS_OBJECTS` because `PREVIEW` "inherits" from `TOUCH_PARIS_OBJECTS` (see the clause at the beginning of the class `PREVIEW` definition). More about this in a future assignment.

- Similarly, it is possible to use results of queries as arguments, e.g. `Console.show (Line8.south_end)`
- The result of an arithmetic expression (say $x \times 3 + 72$) is also an implicit object on which you can call features, e.g. $(x \times 3 + 72).\text{out}$

Expressions built using the "." notation are evaluated from left to right, e.g. $x.y.z.f$ is evaluated as $((x.y).z).f$. This knowledge helps us dissecting feature call instructions.

**Example 1**

```
\begin{array}{ll}
\text{target} & \quad \text{command} & \quad \text{arguments} \\
\text{Line2}.\text{color} & \quad \text{set_red} & \quad \text{Line8}.\text{color}.\text{red} \\
\text{TRAFFIC_LINE} & \quad \text{TRAFFIC_COLOR} & \quad \text{INTEGER} \\
\end{array}
```

Explanation:
- `Line2` is a query defined in class `TOUCH_PARIS_OBJECTS` and returns an object of type `TRAFFIC_LINE`.
- In class `TRAFFIC_LINE` there is a query `color` defined that returns an object of type `TRAFFIC_COLOR`.
- In class `TRAFFIC_COLOR` there is a command `set_red` defined. It takes an argument of type `INTEGER`.
- `Line8` is a query defined in class `TOUCH_PARIS_OBJECTS` and returns an object of type `TRAFFIC_LINE`.
- In class `TRAFFIC_LINE` there is a query `color` defined that returns an object of type `TRAFFIC_COLOR`.
- In class `TRAFFIC_COLOR` there is a query `red` defined that returns an object of type `INTEGER`.
- The argument is thus an `INTEGER` that conforms to the type requested by `set_red`.

**Example 2**

```
\begin{array}{llll}
\text{target} & \quad \text{command} & \quad \text{arguments} \\
\text{Route1}.\text{first}.\text{destination}.\text{stop}(\text{Line8}).\text{line}.\text{south}.\text{end}.\text{highlight} \\
\text{TRAFFIC_ROUTE} & \quad \text{TRAFFIC_LINE} & \quad \text{TRAFFIC_STOP} & \quad \text{TRAFFIC_LINE} & \quad \text{TRAFFIC_STATION} \\
\text{TRAFFIC_LEG} & \quad \text{TRAFFIC_LINE} & \quad \text{TRAFFIC_STOP} & \quad \text{TRAFFIC_LINE} & \quad \text{TRAFFIC_STATION} \\
\end{array}
```

Explanation:
• Route1 is a query defined in class TOUCH_PARIS_OBJECTS and returns an object of type TRAFFIC_ROUTE.

• In class TRAFFIC_ROUTE there is a query first defined that returns an object of type TRAFFIC_LEG.

• In class TRAFFIC_LEG there is a query destination defined that returns an object of type TRAFFIC_STATION.

• In class TRAFFIC_STATION there is a query stop defined that returns an object of type TRAFFIC_STOP and takes an object of type TRAFFIC_LINE as argument.

• Line8 is a TRAFFIC_LINE and thus can be used as such an argument.

• In class TRAFFIC_STOP there is a query line that returns a TRAFFIC_LINE.

• In class TRAFFIC_LINE there is a query south_end that returns a TRAFFIC_STATION.

• And in class TRAFFIC_STATION the command highlight is defined and thus can be called on the target.

A remark on methodology

Generally, long chains of feature calls (with a lot of dots) are considered bad practice, because they tend to be difficult to read and to debug. We include this task to show you how to read these feature calls properly.

To do

For each of the instructions below, determine the type of the target following the scheme from the examples. You will need to read class declarations, so start EiffelStudio and open the project located under traffic/example/02_objects.

Note that for certain classes there exist aliases. As an example, DOUBLE might appear named as REAL_64 and STRING as STRING_8 depending on the view you are using to look at the classes in EiffelStudio.

1. Route2.first.line.extend (Line7.a.i_th (1)) where Route2 is of type TRAFFIC_ROUTE and Line7.a of type TRAFFIC_LINE.

2. Route1.first.next.origin.location.left_by (20.0) where Route1 is of type TRAFFIC_ROUTE.

3. Line2.i_th (Line2.count).stop (Route3.first.line).station.highlight where Route3 is of type TRAFFIC_ROUTE and Line2 of type TRAFFIC_LINE.

Hint

To navigate between classes and features in EiffelStudio, you can use the ‘pick-and-drop’ technique. Just ‘pick’ a class (or a feature) by holding down the [SHIFT] key and right-clicking on the class (feature) name. The cursor will change shape to an oval (or a thick cross in case you picked a feature). You can then ‘drop’ it in another tools pane within EiffelStudio by right-clicking again. When this is not possible, a thin black cross appears on the cursor.

To hand in

Your answers to questions 1-3.
4 Writing more feature calls

To do

1. Download http://se.inf.ethz.ch/teaching/2009-H/eprog-0001/exercises/assignment_3.zip and extract it in traffic/example. You should now have a new directory traffic/example/assignment_3 with assignment_3.ecf directly in it. It is important that the location corresponds to the description here!

2. Open and compile this new project.

3. Open the class text of PLANNER which you will change in this task. Assume that you are planning to change the original metro system of Paris (see Figure 1(a)) in the following way: Line1, Line3, Line8, and Line7_a all only consist of one connection going from the original starting terminal (terminal_1) to Concorde (Station_Concorde). Note that remove_all_segments removes all stations except the terminal 1. Line2 is a cyclic line containing its original terminal 1 and the starting terminal stations of Line3, Line7_a, Line1, and Line8 connected as shown in Figure 1(b).

Hint

To complete the task you need features from the class TRAFFIC_LINE such as remove_all_segments and extend.

In the text editor, when you type the name of an entity followed by a dot, EiffelStudio will automatically display a list of all the features that can be called at the current position (see Figure below). To get the list of almost all features applicable to the Current object, press [CTRL] + [SPACE]. But if you really want to see all the features applicable to the Current object you have to change an option: from the menu Tools/Preferences... choose the directory Editor/Eiffel. Set the ‘Show ANY features’ option to True, and when pressing [CTRL] + [SPACE] you should be able to see, in addition to the others, the most general features, those that can be applied to all objects. Pressing [SHIFT] at the same time will do the same for class names, for example when declaring an attribute or a return type.

To hand in

Submit class PLANNER to your assistant.
Figure 1: Changing the metro system