Assignment 3: Of objects and features

ETH Zurich

Hand-out: 5 October 2007
Due: 12 October 2007

Calvin and Hobbes© Bill Watterson

Goals

• Understand the difference between a class and an object.
• Know a second (professional) way to distinguish between queries and commands.
• Learn to read feature call instructions.
• Write more feature call instructions.

1 Classes vs. objects

To do

1.1 Describe in your own words the difference between a class and an object.
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 (no guessing any more)

There are two main categories of features: queries and commands. In last week’s assignment you categorized features by the sound of their name. This is not a very professional way to do this. In this week’s lecture you learnt how features can be categorized professionally by looking
at their declaration in the class interface. The general patterns for queries and commands in a class interface are:

<table>
<thead>
<tr>
<th>no arguments</th>
<th>with arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>command</strong></td>
<td><strong>feature signature</strong></td>
</tr>
<tr>
<td><strong>command_name</strong></td>
<td><strong>command_name (arg1 : TYPE1; ...; argn : TYPEn)</strong></td>
</tr>
<tr>
<td><strong>query</strong></td>
<td><strong>feature signature</strong></td>
</tr>
<tr>
<td><strong>query_name : TYPE</strong></td>
<td><strong>query_name (arg1 : TYPE1; ...; argn : TYPEn) : TYPE</strong></td>
</tr>
</tbody>
</table>

TYPE, TYPE1, TYPEn are class names. In the case of an argument declaration, it tells you the type of expected arguments. In the case of a query it denotes the type of the object you get as an answer when calling the query. Note 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).

**Todo**

In Listing 1 you find the class interface of `TRAFFIC_TIME` that is responsible in traffic for simulating time in the city (measured e.g. 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.

### 3 Feature reading

In task 2 you saw that feature declarations of queries **always** include the declaration of a return type (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 even clearer:

- Queries return a value (an object), e.g. `Place_balar.position` yields an object of type `TRAFFIC_COORDINATE`, the position of Balard. Since the result is an object, it is possible to apply features to it, e.g. `Place_balar.position.up_by(5.0)`. What features can be applied is defined in the class `TRAFFIC_COORDINATE`. (By the way, `Place_balar` is also a query returning an object of type `TRAFFIC_PLACE` and is declared in class `TOUCH_PARIS_OBJECTS` (why class `PREVIEW` offers this feature is part of the magic and will be resolved later)).

- Similarly, it is possible to use results of queries as arguments, e.g. `Console.show(Line8.terminal_1)`

- The result of an arithmetic expression (say \(x * 3 + 72\)) is also an object on which you can call features, e.g. \((x * 3 + 72).out\)

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 (note that in feature call instructions below the prefix `TRAFFIC` is shortened to `T`).
Example 1

```
target  command  arguments
\underline{Line2.color}. set_red (\underline{Line8.color.blue})
T.LINE    T.LINE
T.COLOR    T.COLOR
```

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 `blue` 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

```
\underline{Route1.first}. destination.stop(\underline{Line8.line.terminal1.highlight})
T.ROUTE   T.LINE
T.ROUTE_SECTION
T.PLACE
T.STOP
T.LINE
T.PLACE
```

Explanation:
- `Route1` is a query defined in class `TOUCH_PARIS_OBJECTS` and returns an object of type `TRAFFIC_PATH`.
- In class `TRAFFIC_PATH` there is a query `first` defined that returns an object of type `TRAFFIC_PATH_SECTION`.
- In class `TRAFFIC_PATH_SECTION` there is a query `destination` defined that returns an object of type `TRAFFIC_PLACE`.
- In class `TRAFFIC_PLACE` there is a query `stop` defined that returns an object of type `T_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`.
- And in class `TRAFFIC_LINE` the command `highlight` is defined and thus can be called on the target.

**To do**

For each statement below, write down its return type and which of the following features could take it as an argument: `put_string (s: STRING)`, `set_length (d: DOUBLE)`, or `set_state (b: BOOLEAN)`. You will need to read class declarations, so open the Preview example in EiffelStudio.

**Example**

Question: `Line8.is_terminal (Place_balard)` where `Line8` is of type `TRAFFIC_LINE` and `Place_balard` of type `TRAFFIC_PLACE`.

Answer: `BOOLEAN`, `set_state`

1. `Michela.intended_line.item.is_exchange` where `Michela` is of type `TRAFFIC_PASSENGER`.
2. `Line8.terminal_1.stop` (Line2).`out` where `Line8` and `Line2` are of type `TRAFFIC_LINE`.
3. `Route1.first.next.next.origin_dummy_node.position.x` where `Route1` is of type `TRAFFIC_PATH`.
4. `Route1.first.type.is_equal (Michela.intended_line.type)` where `Route1` is of type `TRAFFIC_PATH` and `Michela` of type `TRAFFIC_PASSENGER`.
5. `Line2.i_th (Line2.count).stop` (Michela.intended_line).`line.connection_count.to_double` where `Line2` is of type `TRAFFIC_LINE` and `Michela` of type `TRAFFIC_PASSENGER`.

**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 feature/class name) and ‘drop’ it in another pane within EiffelStudio, and see what happens.

**To hand in**

Your answers to questions 1-5.

**4 Writing more feature calls**

**To do**

1. Download `http://se.inf.ethz.ch/teaching/2007-F/eprog-0001/exercises/assignment_3.zip` and extract it in `traffic/example`. You should now have a new directory `traffic/example/ev_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) in the following way: Line1, Line3, Line8, and Line7_a all only consist of one connection going from the original start terminal (`terminal_1`) to Concorde. Line2 is a cyclic line connecting these terminals. The final metro system should look as in Figure 2.
Figure 1: original metro system

Hint
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 3). 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.../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.

To hand in
Submit class PLANNER to your assistant.
Figure 2: new metro system

```
do
    make_scene_default
    traffic_map := a_traffic_map
    number_count := a_number_count
    build_map
end
```

Figure 3: Intellisense
Listing 1: Class `TRAFFIC_TIME`

```plaintext
defered class interface `TRAFFIC_TIME`

3 feature -- All features

5    pause -- Pause the time count.
7        require is_time_running
9        ensure not is_time_running
11    actual_time: TIME
13        -- Simulated time
15    reset -- Reset the time to (0:0:0).
17        ensure
19            is_time_running = False
20            actual_time.hour = 0
21            actual_time.minute = 0
22            actual_time.second = 0
23    duration (a_start_time, a_end_time: like actual_time): TIME_DURATION
25        -- Duration from ‘a_start_time’ until ‘a_end_time’.
27        require both_exist: a_start_time /= Void and a_end_time /= Void
29        ensure
31            result_exists: Result /= Void
32            result_positive: Result.is_positive
34    speedup: INTEGER_32
36        -- Speedup to let the time run faster than the real time
38    set_speedup (a_speedup: INTEGER_32)
40        require a_speedup_valid: a_speedup >= 1
42        ensure
44            speedup_set: speedup = a_speedup
46    start
48        -- Start to count the time at (0:0:0).
50        require not is_time_running
52        ensure
54            is_time_running
56    is_time_running: BOOLEAN
58        -- Is the time running?
60    resume
62        -- Resume the paused time.
64        require not is_time_running
66        ensure
68            is_time_running
70    set (a_hour, a_minute, a_second: INTEGER_32)
72        -- Sets the time to (`a_hour':'a_minute':'a_second').
74        require valid_time: a_hour >= 0 and a_minute >= 0 and a_second >= 0
76    invariant
78            actual_time.hour >= 0
79            actual_time.minute >= 0
81 end
```