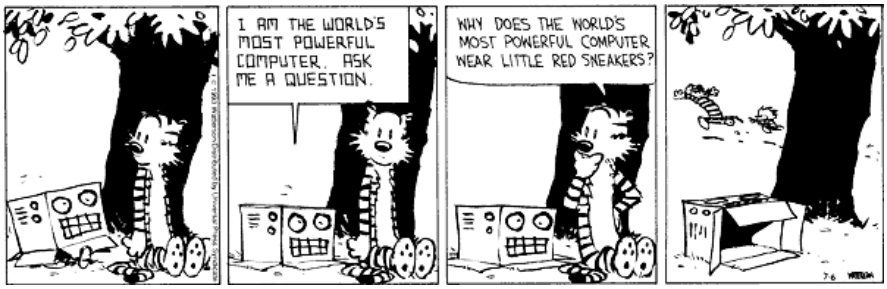
Assignment 2: Give me your feature name and I'll call you

ETH Zurich

Hand-out: 28 September 2009
Due: 6 October 2009



Calvin and Hobbes© Bill Watterson

Goals

- Write more feature calls.
- Get used to EiffelStudio (editor and debugger) .
- Know a first way to distinguish between queries and commands.
- Learn what makes up a valid feature call.

1 Adding more feature calls

Open the 02_objects system again and open the class *PREVIEW* in the editor area.

Todo

1. In feature *explore*, add an additional feature call to revert the actions of highlighting Line8 (use feature *unhighlight*). As the highlight-unhighlight sequence will probably happen too fast for your eyes to notice, put a *wait* instruction between the two and execute the system again. The feature *wait* halts the application for a couple of seconds and updates the screen. Notice that *wait* is not invoked as the other features, (by using an object name and then a dot), but just as it is. This is a shortcut for features that are defined in the same class or in an ancestor class. A call "without a dot" is called an "unqualified call". A call "with a dot" is called a "qualified call". An ancestor class for a class C is a class from which C inherits from. You may have noticed the "inherit" clause after *class PREVIEW*. For the moment think about inheritance as a way to reuse code. In fact you can have different classes that may inherit from *TOURISM* to take advantage of its features.

2. Now that you know the drill, do the same with the Louvre (feature *unspotlight*). Execute the system again.

3. Let's now look for feature *wait*. Where is it? We now know that as it appears in an unqualified call, it must be defined either in the same class or in an ancestor class. In *PREVIEW* there is no *wait*, so how can we check the ancestors? At the beginning of the class you can see that *PREVIEW* inherits from *TOURISM*, so it makes sense to look into *TOURISM*. Right-click on the label *TOURISM* and choose the option "Retarget to class TOURISM". This will bring class *TOURISM* into the editor area. You can also type "TOURISM" in the drop down box on the top left (labeled "Class"), if you wish. Let's now check the features of class *TOURISM*. On the bottom of the right panel (labeled "Groups") select the tab labeled "Features". You should now see a synthetic view of all the features of class *TOURISM*. Unfortunately, *wait* is not there yet, but we still have hope. *TOURISM* inherits from *TOUCH_PARIS_OBJECTS*, so you can repeat what you have just done and finally you should find the wanted feature. This is instructive the first time you do it, but it will become boring when you have to do it again and again, so here is a shortcut. While in class *PREVIEW*, find the drop down box labeled "Feature", and type "wait" in it. Alternatively, right click on *wait* in the program text and then select "Retarget to Feature wait". This will bring up the desired feature in class *TOUCH_PARIS_OBJECTS*.

4. Add additional feature calls to the end of feature *explore*:

   i Display information about Line2 on the Console by using feature *out* and *wait*.

   ii Highlight Line2 and *wait*.

   iii Highlight the south end of Line2 (using features *south_end* and *highlight*) and *wait*.

   iv Highlight the north end of Line2 (using features *north_end* and *highlight*) and *wait*.

5. Until now you have compiled and executed a program without having the possibility to check what was happening every time each single instructions was executed. In fact, to be able to see the effect of certain instructions, you had to insert some *wait* instructions.

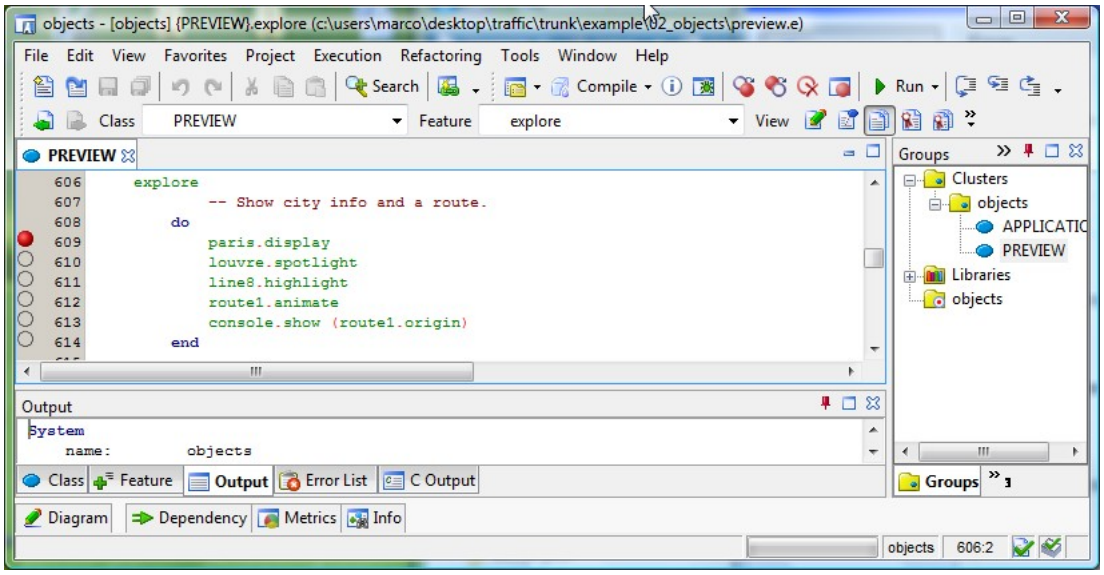
   Now let's see how to use EiffelStudio in "debug mode". "Debugging" means to eliminate bugs. This is a colorful term to refer to defects. A defect is a property of a software system that may cause the system to depart from its intended behavior. Being in debug mode means being able to observe the application execution instruction by instruction, therefore increasing the chances to discover bugs.

   In EiffelStudio you have different views on the code: the basic text view, the clickable view, the flat view, the contract view and the interface view. For the moment we are only interested in the basic text view and the flat view. The basic text view is the view you are using to write code. The flat view lets you see all the features of a class, including the ones coming from the ancestors. You can switch between the views by clicking on the corresponding buttons located to the right of the feature search box mentioned earlier.

   After having selected class *PREVIEW*, try clicking on the flat view button. Then scroll down with the right side bar until you find an old friend: feature *wait*.

   You may have noticed that there are some gray circles at the left of the editor area. These circles identify instructions that will be executed. Look for the feature *explore* and click on the gray circle to the left of the instruction *paris.display*. It should become red. You have just set a breakpoint, in which the program will stop execution.

   Now click on the green full arrow and the program will start. After clicking on "Run example" as usual, the program will stop its execution at your breakpoint. Now you can observe the program behavior step by step by clicking on the button at the top, at the right of the one with a red square. Find it: by hovering over it with the mouse pointer



EiffelStudio in debug mode

you should see the tooltip: "Execute the execution one step at the time", which sounds funny, but it is what it does.

Finally, click on the button once and observe that the first instruction gets executed. Note the little green arrow inside the circle, that points you to the next instruction that will be executed.

Continue the step by step execution until you are done with the features in *explore*. Resume the normal execution by clicking on the green full arrow again.

To hand in

Hand in the code of feature *explore*.

2 Command or Query?

Todo

The features listed below can be found in class *TRAFFIC_STATION*, representing stations in the city to which roads may lead or that may have public transportation lines passing through. To introduce an important distinction between features, we want to find out which features are commands and which features are queries. If you look at the feature name, you may have a hint. You will learn more about this on the next assignment, but for the moment let's have a look at the feature definition. If it appears in the form:

feature_name: CLASS_NAME or *feature_name (...): CLASS_NAME*,

then it is a query. If it appears in the form:

feature_name or *feature_name (...)*,

then it is a command.

Now that you know the trick, for each of the following features in *TRAFFIC_STATION*, figure out whether it is a command or a query:

1. Feature *is_exchange*, like in *Station_balard.is_exchange*.
2. Feature *set_location*, like in *Station_balard.set_location (a_point)*.
3. Feature *outgoing_line_connections*, like in *Station_balard.outgoing_line_connections*.
4. Feature *name*, like in *Station_balard.name*.
5. Feature *highlight*, like in *Station_balard.highlight*.
6. Feature *has_stop*, like in *Station_balard.has_stop (Line7)*.

To hand in

Submit your solution to your assistant.

3 Valid feature call instructions

A feature call instruction is composed of an optional *target* (the object to which an operation is applied), exactly one *command* (the operation to apply), and possibly some arguments. We observe the following:

- The target and the arguments are expressions made up by queries only.
- No commands are allowed as arguments or target.
- Queries may have arguments themselves (see the fifth example below).
- There is exactly one command in a feature call instruction. It appears after the optional target and may be followed only by its arguments.

Below you find examples of feature call instructions where queries are marked in light yellow and commands are marked in dark red. The first six instructions are valid (i.e. they compile) and the others are invalid instructions. For the invalid instructions an explanation is given in square brackets. Make sure you understand these examples.

√ *Station_Balard . highlight*
√ *Line1 . south_end . location . left_by (Line1 . south_end . width)*
√ *Line7_a . set_color (Line3 . color)*
√ *wait*
√ *Paris . station_at_location (Station_Balard . location). unhighlight*
√ *Console . show (Line3 . south_end . has_stop (Line7_a))*
χ *Station_Balard . is_highlighted* [no command in instruction]
χ *Paris . station_at_location (Station_Balard . unhighlight)* [command in argument]
χ *Line7_a . set_color (Line3 . set_color (Line8 . color))* [command in argument]
χ *Line1* [no command in instruction]
χ *Console . show (Line3). Station_Balard* [query after command]

Todo

Assume that *highlight*, *show*, *set_red*, *set_color*, and *set_location* are commands and all other feature names denote queries. Which of the following instructions are valid? Explain your decision. Note that you do not need your computer to answer these questions!

1. *Console.show.Station_Balard*
2. *Station_Balard.set_location (Station_Issy.location)*
3. *Line2.set_color (Line8.highlight)*
4. *Line2.color.set_red (Line8.color.red)*
5. *Console.show (Paris.station_at_location (Station_Balard.location))*
6. *Console.show (Paris.station_at_location (Station_Balard.location).name)*
7. *Line8.north_end.set_location (Route1.city.station_at_location (Station_Balard.location).set_location)*

To hand in

Submit your answers to your assistant.