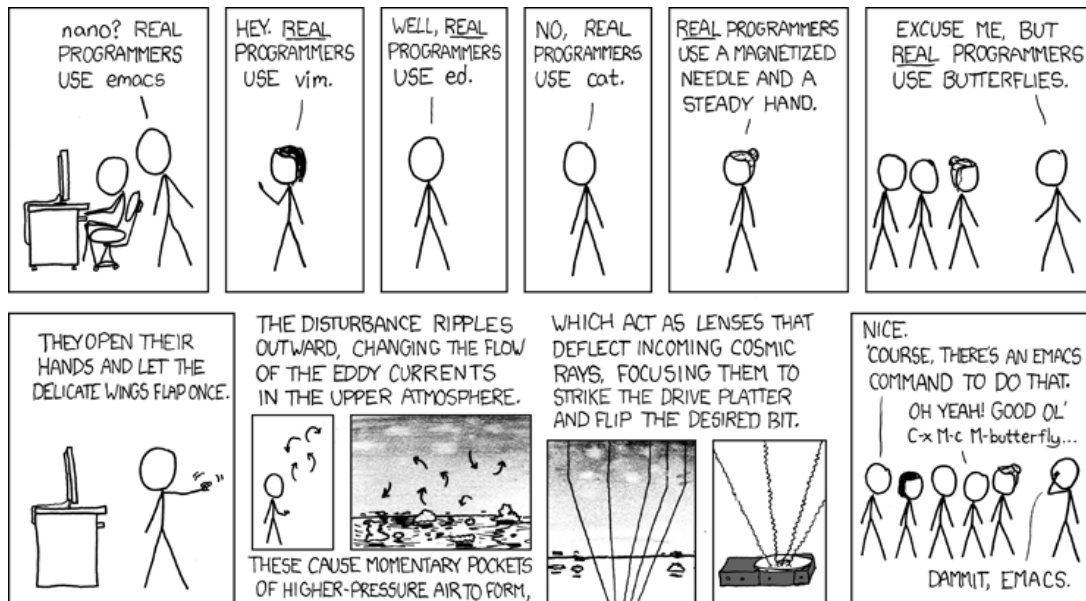# Assignment 2: Give me your feature name and I'll call you

ETH Zurich

Hand-out: Monday, 24 September 2012
Due: Wednesday, 3 October 2012

Real Programmers © Randall Munroe (http://xkcd.com/378)

Goals

- Write more feature calls.

- Write your first standalone program.

- Get used to EiffelStudio (editor, navigation and debugger).

- Learn to distinguish between queries and commands.

- Learn what makes up a valid feature call.

1 Zurich needs more stations

In this task you will continue exploring Traffic and write more feature calls, with and without arguments.

To do

1. Download http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/02/traffic.zip
   and unzip it to a folder of your choice. Open `assignment_2.ecf` in EiffelStudio and look
   at the class *PREVIEW* again: you will see that the feature *explore* is now empty. If you
   run the program, you will see the map of Zurich and nothing else going on (nothing is
   highlighted or animated). In this task you will add feature call instructions to *explore*
   (between **do** and **end**) and check how they affect the map.

2. Let us add a new station to one of the tram lines in Zurich. To achieve that you will have
   to call a feature on the predefined object *Zurich*. Try to find out which feature it is: type
   Zurich followed by a dot, and go through the pop-up list until you see something that
   fits. Note that this feature requires some arguments: you have to provide Traffic with the
   name for the new station as well as its x and y coordinates. The name can be any text
   string you like in double quotes, for example **"Zoo"**. The coordinates are measured in
   meters starting from the city center. For example, $x = 1800$ and $y = -500$ would denote
   a location 1800 meters to the east and 500 meters to the south from the city center.

3. To add the new station to one of the existing tram lines invoke the feature *connect_station*
   on *Zurich*, giving it as arguments the line number and the name of the station.

4. If you run the program now, you won't see any changes on the map. This is because in
   Traffic we distinguish two kinds of objects: the model of the city (in our case *Zurich*) and
   its visual representation, the view (in our case called *Zurich_map*). Whenever you change
   the model, you have to let the view know that it has to update itself accordingly. Add a
   call to *Zurich_map.update* and you will see that the new station now appears on the map.
   If you would like the map to zoom automatically so that the new station fits on the screen,
   add a call to *Zurich_map.fit_to_window*.

5. Now let us attract some attention to the new station and make it blink. You can achieve
   that by highlighting and unhighlighting the view of the station several times in a row. In
   order to access the station view use the expression *Zurich_map.station_view* (...), which
   takes a station as an argument; to get to the station you can use *Zurich.station* (...), and
   provide the station name as an argument, for example **"Zoo"**.

   To make the blinking visible, call the feature *wait* after each highlight and unhighlight
   instruction, giving as an argument the number of seconds you want to wait. Notice that
   wait is not invoked as the other features, by using an object name and then a dot, but
   just as it is (it is an *unqualified call* [Touch Of Class, page 134]).

6. Let us find out where the feature *wait* comes from. As it appears in an unqualified call, it
   must be defined either in the same class or in an *ancestor* class. An ancestor class for a class
   C is a class that *C inherits* from. You may have noticed the **inherit** *ZURICH_OBJECTS*
   clause after **class** *PREVIEW*. It means that *PREVIEW* can use all the features defined
   in *ZURICH_OBJECTS*.

   In *PREVIEW* there is no *wait*, so let us check *ZURICH_OBJECTS*. Right-click on the
   label *ZURICH_OBJECTS* and choose the option "Retarget to class *ZURICH_OBJECTS*".
   You can also type "zurich objects" in the drop down box on the top left (labeled "Class").

   Let us now check the features of class *ZURICH_OBJECTS*. On the bottom of the right
   panel select the tab labeled "Features". You should now see a list of all the features defined
   in class *ZURICH_OBJECTS*, including *wait*.

   Hint There are two shorter ways to find *wait*. While in class *PREVIEW*, type "wait" in
   the drop down box labeled "Feature" above the editor window. 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 *ZURICH_OBJECTS*.

Hint To check the list of all available features, press "Ctrl + Space" while in the editor window. To check the list of available features, whose names start with a certain prefix, type this prefix and then press "Ctrl + Space" (see Figure 1).
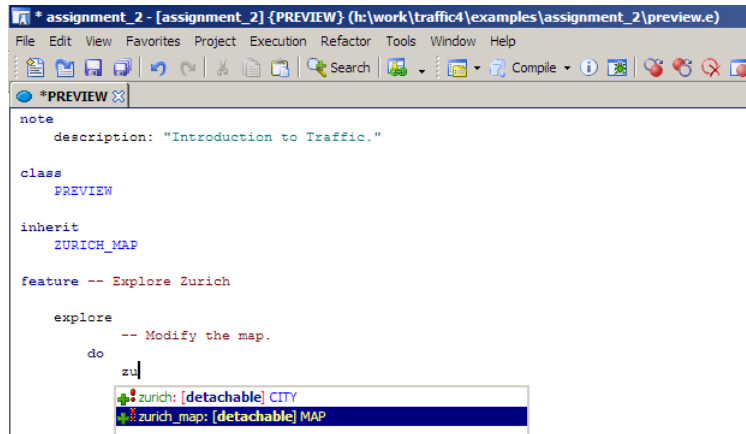

Figure 1: EiffelStudio's auto-completion feature

To hand in

Hand in the code of feature *explore*.

2 Introducing yourself

In this task you will write your first standalone program (not based on Traffic). The program will introduce yourself to your assistant.

To do

1. Download http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/02/introduction.zip and unzip it to a directory of your choice.

   Open "introduction.ecf" in EiffelStudio. In the "Groups" tool on the right you can see that the whole project consists of a single class *APPLICATION*. Open this class in the editor. You will see that it has a single feature, *execute*, whose body is empty so far.

2. Modify the feature *execute* so that it prints the following text (replace the information about John Smith with your personal data):

```
Name:  John Smith
Age:  20
Mother tongue:  English
Has a cat:  True
```

   You can also add any other information you like.

   To do the printing you will use the predefined object called *Io* (input-output). The features you can call on *Io* are defined in the class *STD_FILES*. Browse this class to find the features you need. In particular pay attention to:
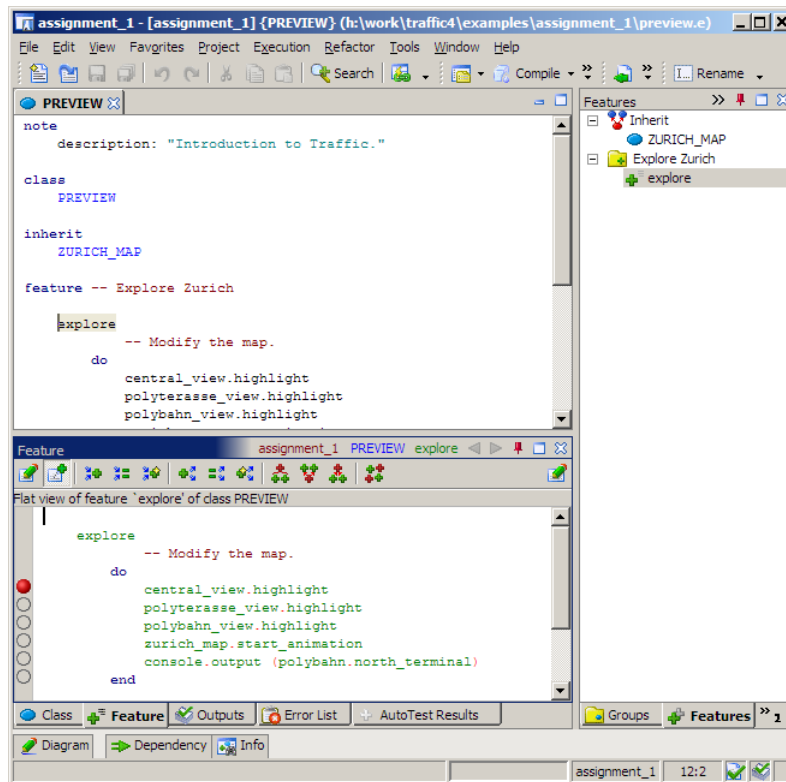
Figure 2: Setting a breakpoint

- feature *put_string* that takes a text string (e.g. **"Hello, world!"**) as an argument and prints it;

- feature *put_integer* that takes an integer number (e.g. 5) as an argument and prints it;

- feature *put_boolean* that takes a boolean value (**True** or **False**) as an argument and prints it;

- feature *new_line* that moves to the next line.

Compile and run your program. **Hint:** the console window with the program output does not automatically pop out on all platforms. If your program appears to be doing nothing, look for a minimized window. On Linux and Mac always start EiffelStudio from a console; then the output will be printed to the same console.

3. Until now you have compiled and executed a program without having the possibility to check what happened after every single instruction was executed. Now let us see how to use EiffelStudio in *debug mode* [Touch Of Class, page 170]. Being in debug mode means being able to observe the application execution instruction by instruction, therefore increasing the chances to discover errors ("bugs").

Right-click on the feature name *execute* in the program text and choose "Pick feature execute". Now right-click in the context tool (the area below the editor). The code of *execute* should now appear in the context tool, with gray circles on the left (see an example in Figure 2). These circles identify instructions that will be executed. Click on the first

gray circle; it should become red. You have just set a breakpoint, at which the program will pause execution.

Now click the "Run" button (see Figure 3) or press "F5": the program will start, but almost immediately it will pause its execution at your breakpoint. Now you can observe the program behavior step by step by clicking the "Step" button (or pressing "F10"). To resume the normal execution click on the "Run" button again (or press "F5").
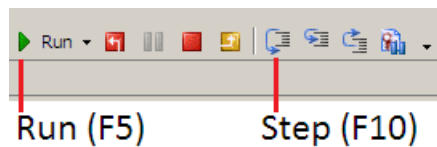

Figure 3: "Run" and "Step" buttons

To hand in

Hand in the code of feature *execute*.

3   Command or Query?

To do

Features listed below can be found in class *CITY*. You task is to find out which features are *commands* and which features are *queries* [Touch Of Class, page 29]. You are supposed to base your decision on the feature *definition* rather then its name (the name can give you a hint, but it can also be misleading sometimes). If the feature definition appears in the class text 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 for each of the following features in *CITY*, figure out whether it is a command or a query:

1. Feature *name*, as in *Zurich.name*.

2. Feature *buildings*, as in *Zurich.buildings*.

3. Feature *add_line*, as in *Zurich.add_line* (2, **"tram"**).

4. Feature *connecting_lines*, as in *Zurich.connecting_lines* (*central*, *polyterrasse*).

5. Feature *move_all*, as in *Zurich.move_all* (0.5).

6. Feature *north*, as in *Zurich.north*.

To hand in

Hand in your answers.