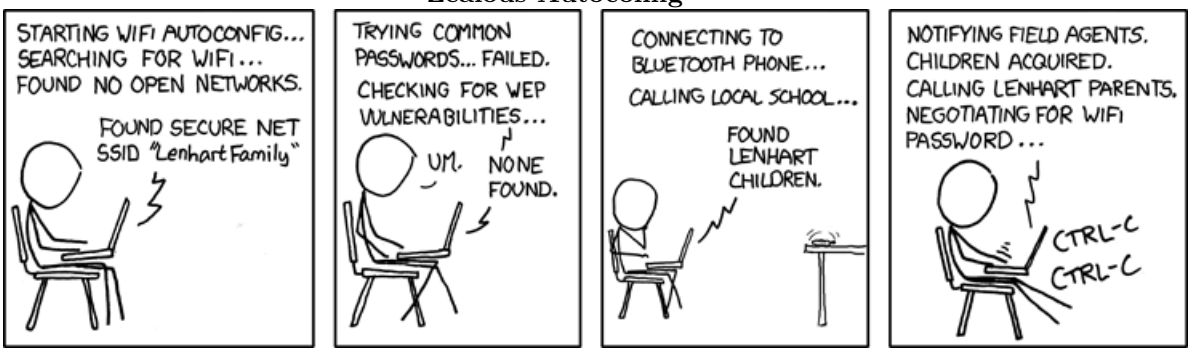
Assignment 5: References and assignments

ETH Zurich

Hand-out: 16 October 2009
Due: 27 October 2009

Zealous Autoconfig



Goals

- Create more objects in Traffic.
- Test your knowledge about assignments.
- Start to work on a more complex application.

1 City building

We have prepared a traffic project that contains a class *CITY_BUILDING*. In this class, you find four features: *explore*, *add_station*, *add_line*, and *random_color*. The application is programmed to call *add_station* when you double click with the left mouse button into the city canvas (the white area where the map is usually displayed), and feature *add_line* when you double click with the right mouse button. At the moment, double clicking will result in a message that is displayed in the Console area of the application, but no station or line is created. In this assignment, you will complete these features to do what their comments promise.

To do

1. Download http://se.inf.ethz.ch/teaching/2009-H/eprog-0001/exercises/assignment_5.zip and extract it in `traffic/example`. You should now have a new directory `traffic/example/assignment_5` with assignment_5.ecf directly in it. It is important that the location corresponds to the description here!

2. Open and compile this new project. Open class *CITY_BUILDING* and follow the suggestions given below.

3. In feature *explore*, create a new object of type *TRAFFIC_CITY*. To let the application know that it has to display this city on the screen, you need to use feature *set_city* of *TRAFFIC_CITY_CANVAS*. You can get the canvas you need from the feature *main_window*. Add a station called "Central station" at coordinate (0, 0) to the city. We have modified the application to automatically call *explore* at startup, but you can still call it by clicking on the "Run example" button.

4. Implement feature *add_line* to add a new line to the city. Use the creation feature *make_with_terminal* of *TRAFFIC_LINE*. The line should be of tram type and have the station that you created in step 3 as south and north end.

5. Implement feature *add_station* to add a new station to the city at the position given through the arguments of *add_station*. The feature should take care of creating a station, add it to the city (a line that already belongs to a city can only include stations that are in the city) and extend the line. The name of the new stations added using this feature should vary according to the order in which they are created: the first additional station created should be "Station 1", the second "Station 2", etc. This means that you should find a way to append consecutive numbers to the fixed string ("Station ") to create each new station name.

6. Extend the line you created last with two new stations. Hint: To make sure that there is a line available you will have to add a call to *add_line* in *explore*.

7. Since all the created lines have the same default color, it is difficult to distinguish them. Implement the feature *random_color* and use it to assign a new color to each created line. To achieve this, you can use a class *RANDOM* that generates random numbers for you. The following code illustrates its usage:

```
local
  t: TIME
  random: RANDOM
  i: INTEGER
  s: INTEGER
do
  create t.make_now                    -- Create a time object for the seed
  s := (t.fine_seconds * 1000).rounded
  create random.set_seed (s)           -- Create the random number generator
                                       -- with 's' as seed
  random.start                         -- think to a list: with 'start' you point
                                       -- to the first valid position
  i := random.item \\ 100              -- Access the first random number.
                                       -- is the modulo operator, used to get a
                                       -- number between 0 and 99
  random.forth                         -- Advance the generator to the next number
end
```

You may have noticed, in the example above, that a seed created from a current time object was used to initialize the *random* object. In general, a seed is used to initialize the algorithm that generates the pseudo-random sequence. Using the same seed means having the same sequence, that's why using time makes sense: running the program at two different moments in time will generate different sequences. Modeling random number

generation with a class that encapsulates the concept of a random sequence is a beautiful example of object-oriented design, and can be also found in other programming languages like Java.

8. In your solution, some entities will probably be local variables, while others are attributes. Think about the criterion you have followed to choose between a local variable and an attribute.

To hand in

Submit the class text of *CITY_BUILDING* to your assistant, and explain the criterion you have followed to choose between local variables and attributes you inserted.

2 Assignments

In this assignment you can test your understanding of assignment instructions. Consider the following class:

```
class PERSON
create make
feature -- Initialization
  make (s: STRING)
      -- Initialize with 's' as 'name'.
    require
      s_exists: s /= Void and then not s.is_empty
    do
      name := s
    ensure
      name_set: name = s
    end

feature -- Access
  name: STRING
  loved_one: PERSON

feature -- Basic operations
  set_loved_one (p: PERSON)
      -- Set 'loved_one' to 'p'.
    do
      loved_one := p
    ensure
      loved_one_set: loved_one = p
    end

invariant
  has_name: name /= Void and then not name.is_empty
end
```

Below is the code of the feature *tryout*. It contains a number of declarations and creation instructions, and it is defined in a class different from *PERSON*. All features of class *PERSON* as shown above are accessible by feature *tryout*.

```
tryout
    -- Tryout assignments
```

```
local
  i, j: INTEGER
  a, b, c: PERSON
do
  create a.make ("Anna")
  create b.make ("Ben")
  create c.make ("Chloe")
  a.set_loved_one (b)
  b.set_loved_one (c)
  -- Here the code snippets from below are added
end
```

To do

You will find a number of subtasks. Each contains a code snippet and statements. Assume that the code snippet is inserted at the location indicated in feature *tryout* above. If the code snippet produces, in your opinion, a compiler error, choose option (a). If it doesn't produce a compiler error, decide for each statement whether it is correct or incorrect after the code snippet has been fully executed. This means that you can have more than one correct statement (provided the compilation went fine!). To make the answers easier to read, we use the short form `Anna` for "the object that has the *STRING* "Anna" as *name* attribute", and accordingly `Ben` and `Chloe` for subtasks 6 – 9.

1.	$j := 3$ $i := j$ $2 := i$	(a) The compiler reports an error. (b) i has value 2, j has value 3. (c) i and j have both value 2. (d) i and j have both value 3.
2.	$i := 7$ $j := 2$ $i := i + 3$	(a) The compiler reports an error. (b) i has value 7 and j has value 2. (c) i has value 5 and j has value 2. (d) i has value 10 and j has value 2.
3.	$i := -7$ $j := 5$ $i := j$ $j := i$	(a) The compiler reports an error. (b) i has value -7 and j has value 5. (c) i and j have both value -7. (d) i and j have both value 5.
4.	$j := 8$ $i := 19$ $j := i$	(a) The compiler reports an error. (b) i and j have both value 19. (c) j has value 19 and i holds no value any more. (d) i and j have both value 8. (e) i has value 8 and j has value 19.
5.	$i := 5$ $j := i + 7$ $i := 8$	(a) The compiler reports an error. (b) i and j have both value 8. (c) i has value 8 and j has value 12. (d) i has value 8 and j has value 15.
6.	$b := a$ $a := b$	(a) The compiler reports an error. (b) a and b are both attached to `Ben`. (c) a is a void reference and b is attached to `Anna`. (d) b is attached to `Anna` and a to `Ben`. (e) a and b are both attached to `Anna`.
7.	$b := a.loved\_one$ $b.set\_loved\_one\ (a.loved\_one)$ $a.set\_loved\_one\ (c)$	(a) The compiler reports an error. (b) The attribute *loved_one* of `Ben` references `Ben`. (c) b is attached to `Chloe`. (d) a is attached to `Anna` and b to `Ben`. (e) b is attached to `Anna` and a to `Chloe`.
8.	$b := c$ $b.loved\_one := a.loved\_one$	(a) The compiler reports an error. (b) b is attached to `Chloe` and its attribute *loved_one* references `Ben`. (c) The attribute *loved_one* of `Chloe` references `Ben`. (d) b is attached to `Ben` and c to `Chloe`.
9.	$b := b.loved\_one.loved\_one$ $a.set\_loved\_one\ (c)$	(a) The compiler reports an error. (b) b is attached to `Chloe`. (c) b is Void and the attribute *loved_one* of a is attached to `Chloe`. (d) a is attached to `Anna` and b to `Ben`. (e) The object with name `Ben` is not reachable any more.

To hand in

Hand in your solution to the questions above.

3 Programming a boardgame: Part 1

To do

In this task you will start a small project from scratch. We will proceed in iterations, starting with a simplified problem and then progressively enriching it. This first part will focus on choosing the right classes.

Problem Description

The idea is to program a prototype of a board-game [1]. It comes with a board, divided into 40 squares, a pair of dice, and can accommodate 2 to 6 players. It works like this: all players start from the first square. One at the time, players take a turn. This includes rolling the dice and advance their respective tokens on the board. When all players are done with their turn, it is called a round. The winner will be the player that first advances beyond the 40th square.

Hints

One of the issues that everyone faces when using an object-oriented language is being able to pick nice abstractions, that is, classes. While there are not rules carved in stone, and granted that experience is of the essence, we can still try to come up with something that makes sense. An advantage of an iterative design process is that we may change our mind later, separate previously united classes or put together previously separated ones.

To suggest classes, you have to ask yourselves:

- What are the relevant abstractions in the problem domain?

The main source can (like in your case here) be a problem description. A class is identified by a name, and the description suggests some names, so it is tempting to use those, isn't it?

Unfortunately the truth is a bit more complicated. The problem is that on the one hand we can find names in the description that may not deserve to be used as a class name (like "idea" or "program"), while on the other hand there may be relevant abstractions that are not expressed as names in the specific text we are looking at.

Still it is worth looking at the names in the text to have some guidance while preparing a first candidate list, and then judge case by case.

In general, what is relevant and what is not depends on the problem domain. If we have to model a door, and its only relevant behavior is that can be open or close, a boolean variable will happily serve the purpose. If you are programming an application in which there can be trapdoors, or magic doors that trigger some non-trivial behavior, then you may need a class for it.

The example above suggests that after having prepared the candidate list, you should go through it again and ask yourself the following question:

- Can I find meaningful data (attributes) and routines that should be included in the class definition?

To hand in

Which classes would you pick to model this problem? Provide your candidate list to your assistant. And please remember that your list is by no means final, so don't be shy and try to include what you really think should be there! In fact for the next part of the exercise we will pick a reasonable list and we will go on, altogether, from there.

[1] We draw inspiration from a case study in the excellent book by Craig Larman: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)