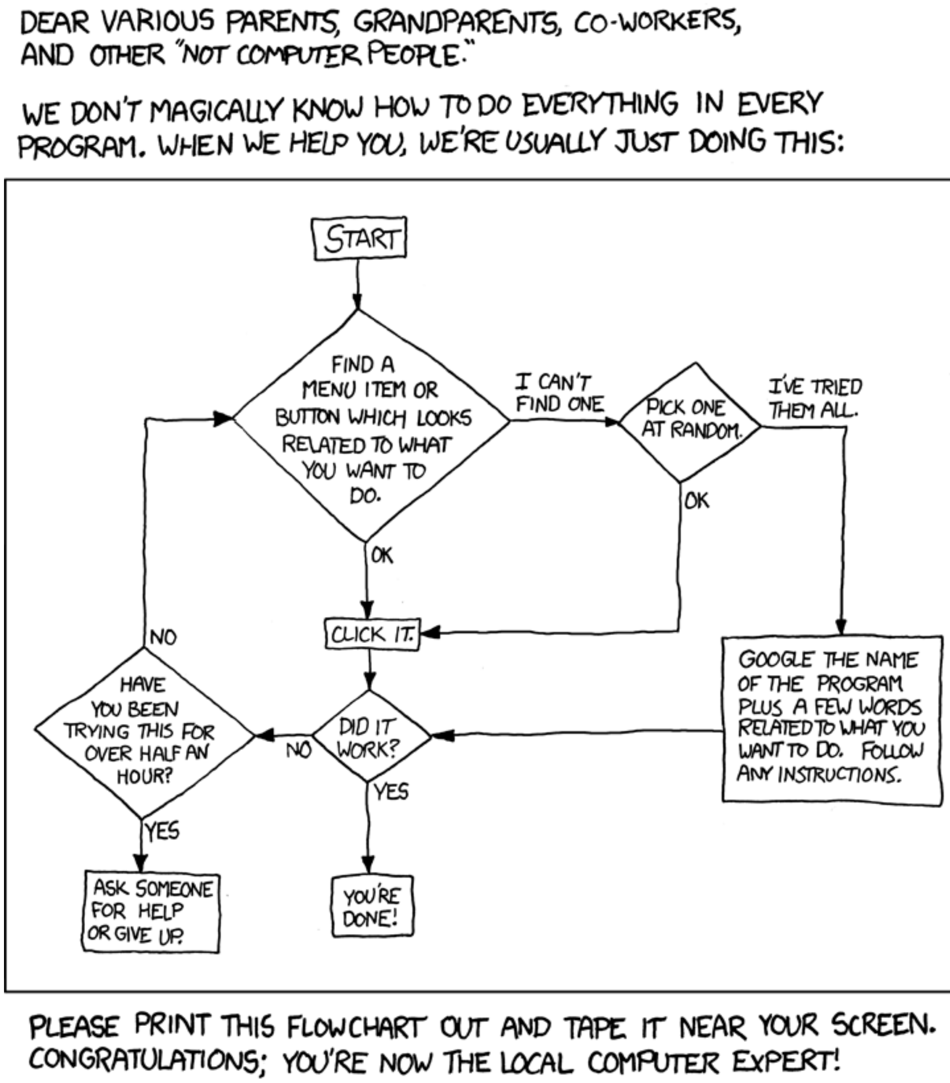
Assignment 6: Loops and conditionals

ETH Zurich

Hand-out: 23 October 2009
Due: 3 November 2009

Tech Support Cheat Sheet


DEAR VARIOUS PARENTS, GRANDPARENTS, CO-WORKERS, AND OTHER "NOT COMPUTER PEOPLE."
WE DON'T MAGICALLY KNOW HOW TO DO EVERYTHING IN EVERY PROGRAM. WHEN WE HELP YOU, WE'RE USUALLY JUST DOING THIS:
START
FIND A MENU ITEM OR BUTTON WHICH LOOKS RELATED TO WHAT YOU WANT TO DO.
I CAN'T FIND ONE
PICK ONE AT RANDOM.
I'VE TRIED THEM ALL.
OK
OK
CLICK IT.
GOOGLE THE NAME OF THE PROGRAM PLUS A FEW WORDS RELATED TO WHAT YOU WANT TO DO. FOLLOW ANY INSTRUCTIONS.
DID IT WORK?
NO
HAVE YOU BEEN TRYING THIS FOR OVER HALF AN HOUR?
NO
YES
YES
ASK SOMEONE FOR HELP OR GIVE UP.
YOU'RE DONE!
PLEASE PRINT THIS FLOWCHART OUT AND TAPE IT NEAR YOUR SCREEN. CONGRATULATIONS; YOU'RE NOW THE LOCAL COMPUTER EXPERT!

Goals

- Understand loops and conditional instructions.
- Use loops and conditional instructions to solve tasks.
- Practice nested loops.

1 Reading loops

What you need to know

Loops theory review

The structure of a loop is shown in Listing 1 and an example of a loop in Eiffel is given in Listing 2. To understand more in detail the meaning of the loop constituent instructions please refer to section 7.5 of Touch of Class.

Listing 1: Loop structure

```
from
    initialization_instructions
invariant
   invariant_clause
until
   exit_condition
loop
   loop_instructions
variant
   variant_clause
end
```

Listing 2: Loop example

```
loop_example
    -- Execute a loop that prints
    -- numbers from 1 to 100.
  local
    count: INTEGER
  do
    from
      count := 1
    invariant
      count >= 1
      count <= 101
    until
      count > 100
    loop
      io.put_integer (count)
      io.put_new_line
      count := count + 1
    variant
      101 - count
    end
  end
```

Comparisons

Operator "=" compares two references. If they are pointing to the same object the result is true, otherwise is false. In contrast, *is_equal* enables to define a personalized implementation of equality. In the case of class *STRING*, this has already been done and does what we expect it to do: compares the two strings character by character. Let's see an example:

Listing 3: *is_equal* vs. =

```
equality_test is
    -- Test string equality.
```

```
local
  s1, s2: STRING
do
   s1 = "abc"
   s2 = "abc"
   if (s1 = s2) then
     print ("s1 = s2: True")
   else
     print ("s1 = s2: False") -- This is printed: different objects
   end
   if (s1.is_equal(s2)) then
     print (" s1.is_equal(s2): True") -- This is printed as expected
   else
     print (" s1.is_equal(s2): False")
   end
   s1 := s2
   print (" After s1 := s2:")
   if (s1 = s2) then
     print (" s1 = s2: True ") -- This is printed: same object
   else
     print (" s1 = s2: False ")
   end
   if (s1.is_equal(s2)) then
     print (" s1.is_equal(s2): True") -- This is printed as expected
   else
     print (" s1.is_equal(s2): False")
   end
end
```

Container operations

- Feature *start* for container objects sets the internal cursor position to the beginning of the list. Imagine a cursor as a marker internal to the container, intended to support all operations on the container by indicating the current position
- Feature *item_for_iteration* returns the object at the cursor position
- Feature *forth* advances the cursor by one position
- Feature *after* is a boolean query that returns **True** if the cursor position is past the last element

It happens very often that you want to iterate through all the items of a container in Traffic (e.g. through *Paris.stations*, *Paris.lines*, or *Paris.passengers*). To do this you can use the following scheme (here for *Paris.lines*, similar for the other containers in a *TRAFFIC_CITY*):

Listing 4: Looping through map item containers

```
from
  Paris.lines.start
until
  Paris.lines.after
loop
  Paris.lines.item_for_iteration.highlight
  Paris.lines.forth
end
```

To do

Assume that the two code extracts in Listing 5 and Listing 6 intend to loop through a list of stations and search for the station named "Cite Universitaire" and highlight it.

1. For each version (Listings 5 and 6) decide whether it does what it is supposed to do.
2. If you think it is not OK, then correct the errors.

Listing 5: Version A

```
explore is
    -- Highlight "Cite Universitaire".
  local
    found: BOOLEAN
  do
    Paris.display
    from
      Paris.stations.start
    until
      Paris.stations.after or found
    loop
      if Paris.stations.item_for_iteration.
          name = "Cite Universitaire"
          then
        found := True
      else
        Paris.stations.forth
      end
      if not Paris.stations.after then
        Paris.stations.item_for_iteration.
            highlight
      end
    end
  end
```

Listing 6: Version B

```
explore is
    -- Highlight "Cite Universitaire".
  do
    Paris.display
    from
      Paris.stations.start
    until
      Paris.stations.after or Paris.stations.
          item_for_iteration.name.is_equal ("
            Cite Universitaire")
    loop

    end
    if not Paris.stations.after then
      Paris.stations.item_for_iteration.
          highlight
    end
  end
```

Hints

For this exercise you may assume the following:

- There are no compilation problems
- All the entities are not Void (i.e. they are all attached to an object)

To hand in

This is a pen-and-paper exercise: you do not need to write code in EiffelStudio. Hand in the corrected versions of Listing 5 and Listing 6.

2 Equipping Paris

In this task you will be exercise loops in the context of the Traffic application.

To do

1. Download http://se.ethz.ch/teaching/2009-H/eprog-0001/exercises/assignment_6.zip and extract it in `traffic/example`. You should now have a new directory `traffic/example/assignment_6` with assignment_6.ecf directly in it (it is important that the location corresponds to the description here!).

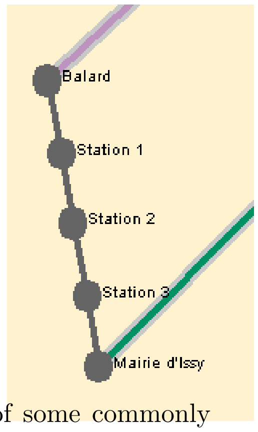
2. Open and compile this new project. Open class *LOOPINGS* and solve the tasks below.

3. Implement the feature *generate_trams_for_line*. This feature has a line as argument and should check whether the line is a tram line. To find out whether a *TRAFFIC_LINE* object is a tram line, you need to create an object of type *TRAFFIC_TYPE_TRAM* and see if the line's attribute *type* is equal to it. If the line is a tram line then the feature should create for every second station a tram that starts moving at this station. So the first tram should start at the first station of the line, the second tram at the third station, the third tram at the fifth station, etc. Use feature *set_to_station (a_station: TRAFFIC_STATION)* to set the initial position of a tram with respect to a certain station (this feature is available in *TRAFFIC_TRAM* because inherits from *TRAFFIC_LINE_VEHICLE*). Don't forget to add the generated trams to *Paris*.

4. Implement feature *equip* to generate trams for all the lines of the city. You may use *generate_trams_for_line* to achieve this.

5. Implement the feature *generate_connecting_bus_line*. The idea of this feature is to create a new bus line with *n* intermediary stops that connects the given *start_station* to the *end_station*. As a first step, you need to create a new line of bus type (use *make_with_terminal* as creation procedure and the argument *start_station* as the terminal station). Then use a loop to create *n* new stations and extend the line with them. After doing so, add the *end_station* to the line. The locations of the intermediary stations should be evenly distributed along the straight line between the *start_station* and the *end_station*. In the example seen in the figure below, *n* was 3, the start station Balard and the end station Mairie d'Issy. To calculate the locations of the newly created stations you need to do some vector calculations based on the locations of the start and end stations. *TRAFFIC_POINT* provides some so called **infix**-features (+, −, ∗) that will help you:

Vector addition	*a, b, c: TRAFFIC_POINT*	c := a + b
Vector subtraction	*a, b, c: TRAFFIC_POINT*	c := a - b
Scalar multiplication	*a, b: TRAFFIC_POINT* *f: DOUBLE*	b := a * f (Note: the scalar needs to be the second operator)
Scalar division	*a, b: TRAFFIC_POINT* *f: DOUBLE*	b := a / f (Note: the scalar needs to be the second operator)



The infix-operators are in this case meant to overload the behavior of some commonly used operators like "+", "-", "*" and "/". We use the term "overloading" because now you can use them not only to operate on numbers but also on *TRAFFIC_POINT* objects.

Another hint is that you may want to perform some rounding on the computed locations (see feature *rounded* in class *DOUBLE*).

6. Test your implementation of *generate_connecting_bus_line* with some stations (e.g. *Station_balard* and *Station_mairie_d_issy*), adding code to feature *equip*. And, once again, don't forget to add the new bus line to Paris!

To hand in

Hand in the code of class *LOOPINGS*.

3 Loop painting

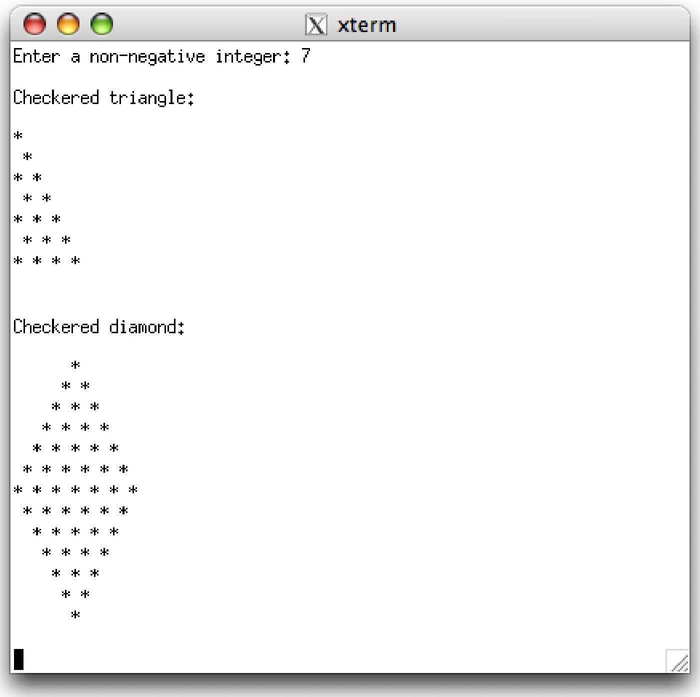
To do

Write a program that does the following:

1. Asks the user to input a positive integer value
2. Displays, using asterisks, a checkered rectangled triangle having as hypotenuse a number of asterisks equal to the input value inserted in step 1. Be aware that stars and white spaces should be alternating
3. Displays, using asterisks, a diamond having as side the same number of asterisks as given in 1. Here as well, the stars and white spaces should be alternating.

Hints

The output should be like in Figure 1. You might need to use the integer division operator // or the modulo operator \\ for your solution.



```
Enter a non-negative integer: 7

Checkered triangle:

*
 *
* *
 * *
* * *
 * * *
* * * *


Checkered diamond:

      *
     * *
    * * *
   * * * *
  * * * * *
 * * * * * *
* * * * * * *
 * * * * * *
  * * * * *
   * * * *
    * * *
     * *
      *
```

Figure 1: Example with value 7

To hand in

Hand in your class text.

4 Programming a boardgame: Part 2

To do

In this task you will implement a given set of classes. They may not coincide with the ones you picked last week, but it is easier to go on altogether in this way. The set of classes you should focus on is the following:

GAME, *DIE*, *PLAYER*, *BOARD*, *SQUARE*.

Also, use class *APPLICATION* as root class of your system. As a reminder, you will find below the description of the problem. It has been slightly modified because it mentions six-sided dice. While this is a little detail, it gives you an idea of the fact that across different iterations of the design and development process the specifications can actually change.

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 six-sided 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

- Which classes need to "know" about the others, and which not? This is important, because if class *A* needs to know about class *B*, then *A* should have an attribute (or may be a local variable, depending on the specific situation) of type *B*. For example, *BOARD* may need to know about *SQUARE*, but not the other way around. You can also assume that players and squares have names like "Player1", "Player2", Square1", "Square2", etc.).
- Who should take care of creating the objects? The answer does not necessarily have to be class *GAME*. Again, *BOARD* and *SQUARE* objects can be good food for thought.
- Class *DIE* can be a nice exercise to check that you have understood random number generation, and the command-query separation principle.
- How is the board built? There are different solutions here. For example, one could be to put in every *SQUARE* object an attribute representing the next square.
- You may want to practice with class *ARRAY* to handle players.

To hand in

Submit the code of classes *GAME*, *DIE*, *PLAYER*, *BOARD*, *SQUARE*.

If you feel lost...

If you tried really hard but you don't have a clue on how to organize the given classes internally, you may want to download the class skeletons (with empty feature bodies) from http://se.ethz.ch/teaching/2009-H/eprog-0001/exercises/boardgame_helper.zip

[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)