

Assignment 6: Smarties

ETH Zurich

Hand-out: 5. December 2005

Due: 13. December 2005

1 Smarter brain

Goal

In this exercise you will improve the bot brain of the estate agent in Flathunt. The estate agent's choice of a next move when played by a bot was not really smart: He always chose the station that was farthest away from his current position. You are now going to change this!

To do

1. First of all you will need to replace some classes so that the estate agent knows about the positions of the flathunters and how the transportation system looks like. Download them (see link below) and replace your files in Flathunt and Traffic with the provided files. Note that the directory provided in the zip will help you to find out where to put the files.
http://se.inf.ethz.ch/teaching/ws2005/0001/exercises/assignment_6.zip
2. How does the estate agent (when played by a bot) choose his next move? This is defined in the feature *choose_next_move* of class *ESTATE_AGENT_BOT*. If you take a look at the feature you will quickly see that so far the estate agent always made the longest possible move. Look at the feature and try to understand how it is implemented.
3. Change the feature *choose_next_move* in class *ESTATE_AGENT_BOT* so that, before taking a move, the estate agent checks what the distance from the flathunters would be if he made that move. The distance is measured in number of line segments between the agent and the nearest flathunter. Of course, the larger the distance the better for the estate agent because he has a bigger chance of escaping. In the end, the agent should therefore choose the move with the largest distance. Check the hints for information on how to calculate the number of line segments between two places (plus other important features).

4. **Optional:** You probably realized that the new strategy, although better than the original one, still is not the smartest solution. Think about the optimal strategy of choosing the next move and implement it.

Hints

- In order to calculate the number of line segments you must add the following feature to the class *ESTATE_AGENT_BOT*.

```

shortest_route (a_location1, a_location2: TRAFFIC_PLACE):
  TRAFFIC_ROUTE is
    -- Calculate the route between two places.
  require
    a_location1_exists : a_location1 /= Void
    a_location2_exists : a_location2 /= Void
    locations_different : a_location1 /= a_location2
  local
    l: LINKED_LIST [TRAFFIC_PLACE]
  do
    create l.make
    l.extend (a_location1)
    l.extend (a_location2)
    create Result.make (l, knowledge.map)
    Result.calculate_shortest_path
  ensure
    Result_exists : Result /= Void
  end

```

- With the feature *knowledge.fathunter_positions* you get access to a list of the positions of all the fathunters.
- To assign the largest possible integer number to a local variable *max_int* you can use the following line of code:

```
max_int := feature {INTEGER}.max_value
```

To hand in

Submit the class text of *ESTATE_AGENT_BOT* to your assistant. Make sure to upload your learning logs! We appreciate your cooperation!

Solution

```

class
  ESTATE_AGENT_BOT

inherit
  BRAIN

feature -- Basic operations

choose_next_move (possible_moves: LINKED_LIST{TRAFFIC_LINE_SECTION};
  my_location: TRAFFIC_PLACE; last_estate_agent_location: TRAFFIC_PLACE) is
  -- Choose next move for the estate agent.
  local
    best_move_so_far: TRAFFIC_LINE_SECTION
    tmp_move: TRAFFIC_LINE_SECTION
    best_distance: DOUBLE
    tmp_distance: INTEGER
    worst_distance: INTEGER
  do
    best_distance := -1
  from
    possible_moves.start
  until
    possible_moves.off
  loop
    from
      knowledge.flathunter_positions.start
      worst_distance := feature {INTEGER}.max_value
    until
      knowledge.flathunter_positions.off
    loop
      if possible_moves.item.destination /= knowledge.flathunter_positions.item then
        tmp_distance := shortest_route (possible_moves.item.destination, knowledge.
          flathunter_positions.item).line_sections.count
      else
        tmp_distance := 0
      end
      if tmp_distance <= worst_distance then
        worst_distance := tmp_distance
      end
      knowledge.flathunter_positions.forth
    end
    if worst_distance > best_distance then
      best_move_so_far := possible_moves.item
      best_distance := worst_distance
    end
    possible_moves.forth
  end
  chosen_move := best_move_so_far
ensure then
  result_exists : chosen_move /= Void
  result_has_place : chosen_move.origin = my_location
end

shortest_route (a_location1, a_location2: TRAFFIC_PLACE): TRAFFIC_ROUTE is
  -- Calculate the route between two places.

```

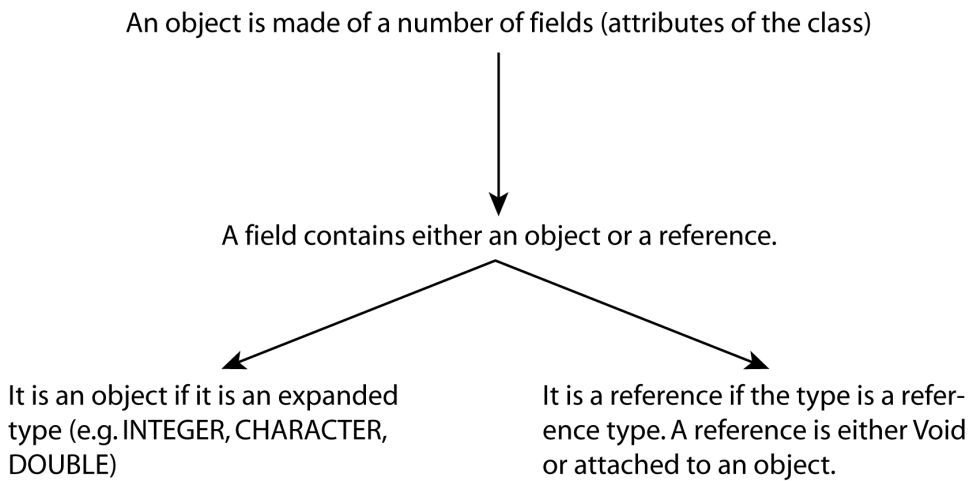
```
require
  a_location1_exists : a_location1 /= Void
  a_location2_exists : a_location2 /= Void
  locations_different : a_location1 /= a_location2
local
  l: LINKED_LIST [TRAFFIC_PLACE]
do
  create l.make
  l.extend (a_location1)
  l.extend (a_location2)
  create Result.make (l, knowledge.map)
  Result.calculate_shortest_path
ensure
  Result_exists : Result /= Void
end
end
```

2 Reference and expanded types, copy and twin

Goal

- Understand the difference between value and reference.
- Understand the difference between deep and shallow.

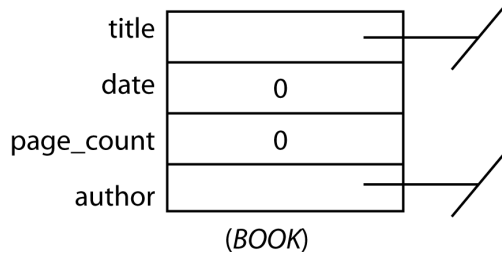
Summary



Example

```
class BOOK
feature
  title : STRING
  date, page_count: INTEGER
```

```
author: WRITER
end
```



Class *BOOK* with 4 fields, 2 of which (*STRING*, *WRITER*) are references.

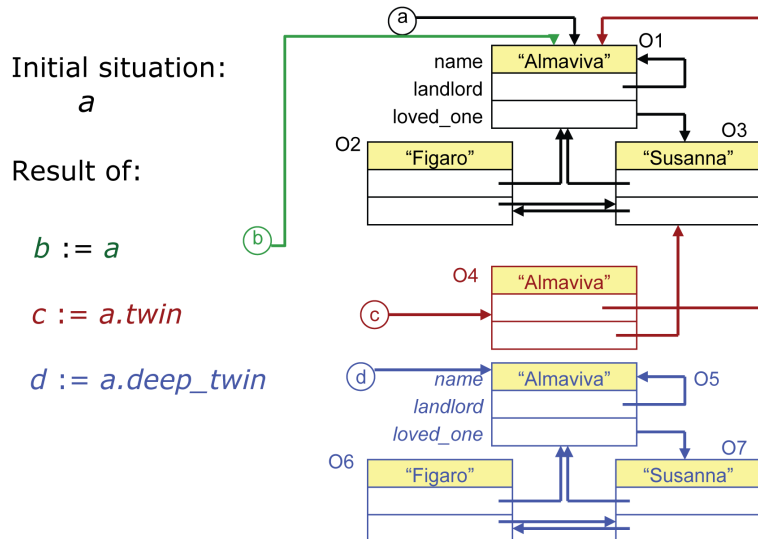
Eiffel offers the following features for reference types:

<code>x = y</code>	compares if two references are attached to the same object
<code>equal (x, y)</code>	compares two objects (shallow comparison)
<code>deep_equal (x, y)</code>	deep comparison of objects x and y
<code>x := y</code>	attaches x to the object denoted by y (reference assignment)
<code>x.copy (y)</code>	copies the content of object y into object referenced by x (shallow copy)
<code>x := y.twin</code>	creates new object attached to x as a shallow copy of y (calls copy)
<code>x := y.deep_twin</code>	creates a new object attached to x as a deep copy of y

and for expanded types:

<code>x = y</code>	compares if two objects are the same
<code>x := y</code>	copies y into x

This is how the object structure before and after some commands looks like:



And here is an analogy:

Consider a HTML-page. You can compare the normal text with objects, and the links with references. Now, if you just save the web page, only the current page will be saved, the links will still point to the same pages in the WWW. However, there are some programs that allow you to make something like a deep copy, where (down to a certain level) also the referenced pages are downloaded, and the links are changed to the local copy of these references pages.

Description

This is a multiple choice exercise. First, make sure that you understood the theory above, and then try to answer questions 1 to 5.

To do

1. Suppose that the instruction $x := y.twin$ has just been executed successfully, which of the following statements are true?
 - (a) `equal (x, y)`
 - (b) $x = y$
 - (c) `deep_equal (x, y)`
2. Suppose that the instruction $x.copy (y)$ has just been executed successfully, which of the following statements are true?
 - (a) `equal (x, y)`

- (b) $x = y$
 - (c) `deep_equal(x, y)`
3. Suppose that the instruction $x := y$ has just been executed, which of the following statements are true?
- (a) `equal(x, y)`
 - (b) $x = y$
 - (c) `deep_equal(x, y)`

Solution

1. Suppose that the instruction `x := y.twin` has just been executed successfully, which of the following statements are true?
 - (a) *equal* (*x*, *y*)
 - (b) `x = y`
 - (c) *deep_equal* (*x*, *y*)
2. Suppose that the instruction `x.copy (y)` has just been executed successfully, which of the following statements are true?
 - (a) *equal* (*x*, *y*)
 - (b) `x = y`
 - (c) *deep_equal* (*x*, *y*)
3. Suppose that the instruction `x := y` has just been executed, which of the following statements are true?
 - (a) *equal* (*x*, *y*)
 - (b) `x = y`
 - (c) *deep_equal* (*x*, *y*)

To hand in

Hand in your multiple choice answers to questions 1 to 4, and the reason for your answer in question 4.

3 BNF-E in BNF-E**To do**

Describe BNF-E with the help of BNF-E. Assume that the lexical constructs **Keyword** and **Symbol** (for terminals) and **Identifier** (for constructs) are given:

```
Terminal ::= Keyword | Symbol
Construct ::= Identifier
```

To hand in

Hand in your solution.

Solution

```
BNF-E ::= {Production "%N" ...}+
Production ::= Construct "==" Definition
Definition ::= Concatenation | Choice | Repetition

Concatenation ::= {Concat_clause " " ...}+
Concat_clause ::= Member | Optional
Optional ::= "[" Member "]"

Choice ::= Choice_clause "|" Member
Choice_clause ::= {Member "|" ...}+

Repetition ::= "{" Construct Member "... }" Mode
Mode ::= "+" | "*"

Member ::= Construct | Terminal
```