

## Solution 6: Loops and conditionals

ETH Zurich

### 1 Reading loops

#### Solution

Concerning Version A:

- First, the result of the comparison using the equality operator `=` will always be **False** for different objects as in this case. In addition to what we have seen in the sample listing we can observe that *STRING* objects are not expanded, and then we use references to manipulate them. In contrast, this is not true when using expanded objects like *INTEGER*.
- Second, the if-statement is inside the loop: the loop will highlight every station until the loop stops.
- The corrected code of version A is shown in Listing 1.

Concerning Version B:

- Endless loop: there is no call to a command that advances the cursor position in the list.
- Possible precondition violation: *stations.item\_for\_iteration.name.is\_equal ("Cite Universitaire")* may be tested before *Paris.stations.after*. In the case where *Paris.stations.after* holds, the call to *Paris.stations.item\_for\_iteration* may violate the precondition *not\_after: not after* of feature *item\_for\_iteration* in class *TRAFFIC\_ITEM\_HASH\_TABLE*. This is because by using *or* instead of *or else* the order of evaluation is not guaranteed.
- The corrected code of version B is shown in Listing 2.

Listing 1: Corrected 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.
        is_equal("Cite Universitaire"
        )) then
      found := True
    else
      Paris.stations.forth
    end
  end
  if (not Paris.stations.after) then
    Paris.stations.item_for_iteration.
    highlight
  end
end
```

Listing 2: Corrected version B

```
explore is
  -- Highlight "Cite Universitaire".
  do
    Paris.display
  from
    Paris.stations.start
  until
    Paris.stations.after or else Paris.
    stations.item_for_iteration.
    name.is_equal("Cite
    Universitaire")
  loop
    Paris.stations.forth
  end
  if (not Paris.stations.after) then
    Paris.stations.item_for_iteration.
    highlight
  end
end
```

## 2 Equipping Paris

### Solution

Listing 3: Class *LOOPINGS*

```
1 indexing
  description: "Loopings class (Assignment 6)"
3
4
5 class
6   LOOPINGS
7 inherit
8
9   TOURISM
10
11 feature -- Explore Paris
12
13   equip
14     -- Build trams and connecting lines.
15   do
16     Paris.display
17     wait
18     from
19     Paris.lines.start
20     until
21     Paris.lines.after
22     loop
23     generate_trams_for_line (Paris.lines.item_for_iteration)
24     Paris.lines.forth
25   end
26   generate_connecting_bus_line (3, station_balard, station_mairie_d_issy)
27 end
28
29 generate_connecting_bus_line (n: INTEGER; start_station, end_station:
30   TRAFFIC_STATION)
31   -- Generate 'n' new stations and a bus line.
32
33   require
34     stations_exist: start_station /= Void and end_station /= Void
35     stations_not_same: start_station /= end_station
36     n_positive: n > 0
37   local
38     l: TRAFFIC_LINE
39     s: TRAFFIC_STATION
40     t: TRAFFIC_TYPE_BUS
41     i: INTEGER
42     v: TRAFFIC_POINT
43   do
44     v := (end_station.location - start_station.location)/(n+1)
45     create t.make
46     create l.make_with_terminal ("Bus line", t, start_station)
47     Paris.put_line (l)
```

```

47     from
48         i := 1
49     until
50         i > n
51     loop
52         create s.make_with_location ("Station " + i.out, ( start_station . location . x + v.x*i ).
53             rounded, ( start_station . location . y + v.y*i ). rounded)
54         Paris.put_station ( s )
55         l.extend ( s )
56         i := i + 1
57     end
58     l.extend ( end_station )
59 end

generate_trams_for_line ( a_line: TRAFFIC_LINE )
60     -- Generate trams for 'a_line' on every second station if allowed.
61     require
62         a_line_exists : a_line /= Void
63     local
64         t: TRAFFIC_TRAM
65         type: TRAFFIC_TYPE_TRAM
66     do
67         create type.make
68         if a_line.type.is_equal ( type ) then
69             from
70                 a_line.start
71             until
72                 a_line.after
73             loop
74                 create t.make_with_line ( a_line )
75                 t.set_to_station ( a_line.item )
76                 t.start
77                 Paris.put_tram ( t )
78                 a_line.forth
79                 if not a_line.after then
80                     a_line.forth
81                 end
82             end
83         end
84     end
85 end
86
87 end
    
```

### 3 Loop painting

#### Solution

Listing 4: Class *DRAWING\_MANAGER*

```

1 class
2     DRAWING_MANAGER
3 create
    
```

```
make
5 feature -- Initialization

7 make
  -- Get size and invoke display.
9 local
  n: INTEGER
11 do
  io.put_string ("Enter a non-negative integer: ")
13 io.read_integer
  n := io.last_integer

15
  io.put_string ("%NChecked triangle:%N%N")
17 print_checker_triangle (n)

19 io.put_new_line
  io.put_new_line

21
  io.put_string ("Checked diamond:%N%N")
23 print_checker_diamond (n)

25 end

27 feature -- Checkerboards

29 print_checker_triangle (n: INTEGER)
  -- Print a checker pyramid of size 'n' by 'n'.
31 local
  i, j, star: INTEGER
33 do
  from
35   i := 1
   star := 0
37 until
   i > n
39 loop
  from
41   j := 1
  until
43   j > i
  loop
45   if j \ 2 = star then
   io.put_character (' ')
47   else
   io.put_character ('*')
49   end
   j := j + 1
51 end
   star := 1 - star
53 i := i + 1
   io.put_new_line
55 end
```

```
57     end
58
59     print_checker_diamond (n: INTEGER)
60     -- Print checker diamond of size 'n'.
61     local
62         i: INTEGER
63         left, middle: STRING
64     do
65         create left.make_filled (' ', n)
66         middle := ""
67         from
68             i := 1
69         until
70             i > n
71         loop
72             left.remove_tail (1)
73             middle.append ("* ")
74             io.put_string (left + middle + "%N")
75             i := i + 1
76         end
77         from
78             i := 1
79         until
80             i > n
81         loop
82             left.append (" ")
83             middle.remove_tail (2)
84             io.put_string (left + middle + "%N")
85             i := i + 1
86         end
87     end
```

## 4 Programming a boardgame: Part 2

### Solution

Note that the implementation chosen reflects what we know about the problem, that is, very little. So we opted for a minimalist solution, trying to avoid code duplication. Notice in particular that class *BOARD* is pretty simple, and only "knows" about the start square. Then the knowledge of who is the next square is delegated to each square. Also check class *DIE*. By using a *once* function *rand* we enforce that only one sequence of pseudo-random numbers will be generated in every application. A worse alternative would be to have *rand* as an attribute, and letting *make* generate the sequence. This would mean that two objects created at a few milliseconds of distance from each other will probably have very similar pseudo-random sequences. This should be likely to result in rolling a lot of doubles (test it yourself).

Listing 5: Class *GAME*

```
1 class
  GAME
3
4 create
5 make
6
7 feature {NONE} -- Initialization
8
9 make
  -- Run application.
11 local
  i: INTEGER
13  p: PLAYER
  do
15    create game_board.make
  create die_1.make
17    create die_2.make
  create players.make (1, number_of_players)
19    from
  i := 1
21    until
  i > players.count
23    loop
  create p.make ("Player" + i.out)
25    p.set_location (game_board.start_square)
  players [i] := p
27    i := i + 1
  end
29 end
30
31 feature -- Basic operations
32
33 play
  -- Start a game.
35 local
  i: INTEGER
37 do
```

```

39     print ("%N*** Simple Boardgame ***")
40     from
41     until
42         has_winner
43     loop
44         from
45             i := 1
46         until
47             has_winner or else i > number_of_players
48     loop
49         players [i].play (die_1, die_2)
50         if players [i].location = Void then
51             print ("%NAnd the winner is: " + players [i].name)
52             has_winner := True
53         end
54         i := i + 1
55     end
56 end
57 print ("%N*** Game Over ***")
58 ensure
59     game_has_winner: has_winner
60 end
61 feature -- Status
62
63     game_board: BOARD
64     -- The game board.
65
66     number_of_players: INTEGER = 2
67     -- For testing purposes, the number of players is set to 2.
68
69     players: ARRAY [PLAYER]
70     -- Container for players.
71
72     die_1: DIE
73     -- The first die.
74
75     die_2: DIE
76     -- The second die.
77
78     has_winner: BOOLEAN
79     -- Does the game have a winner?
80
81 invariant
82
83     game_board_exists: game_board /= Void
84
85     players_exist: players /= Void and then not players.is_empty
86
87     number_of_players_consistent: number_of_players >= 2 and number_of_players <= 6
88
89
    
```

```
91 dice_exist : die_1 /= Void and die_2 /= Void  
end
```

Listing 6: Class *DIE*

```
1 class  
  DIE  
3  
4 create  
5 make  
7  
8 feature {NONE} -- Initialization  
9  
10 make  
11 -- Create a die with valid initial face value.  
12 do  
13   face_value := rand.item \\ 6 + 1  
14 end  
15  
16 feature -- Access  
17  
18   face_value: INTEGER  
19  
20 feature -- Basic operations  
21  
22   roll  
23   -- Roll die  
24 do  
25     rand.forth  
26     face_value := rand.item \\ 6 + 1  
27 end  
28  
29 feature {NONE} -- Implementation  
30  
31 rand: RANDOM  
32 -- Pseudo-random number generator.  
33 local  
34   t: TIME  
35   seed: INTEGER  
36 once  
37   create t.make_now  
38   seed := (t.fine_seconds * 1000).rounded  
39   create Result.set_seed (seed)  
40   Result.start  
41 end  
42  
43  
44 invariant  
45   six_sided_die : face_value >= 1 and face_value <= 6  
46  
47 end
```

Listing 7: Class *PLAYER*

```
1 class
  PLAYER
3
4 create
5   make
6
7   feature -- Access
8
9     name: STRING
11    -- Player name.
12
13   location: SQUARE
14    -- Player current location.
15
16   feature {NONE} -- Initialization
17
18   make (n: STRING)
19    -- Create a player with name.
20   require
21     name_exists: n /= Void and then not n.is_empty
22   do
23     name := n
24   ensure
25     name_set: name = n
26   end
27
28   feature -- Status setting
29
30   set_location (loc: SQUARE)
31    -- Set location for player.
32   require
33     location_exists : loc /= Void
34   do
35     location := loc
36   ensure
37     location_set : location = loc
38   end
39
40   feature -- Basic operations
41
42   play (d1,d2: DIE)
43    -- Play a turn.
44   require
45     dice_exist : d1 /= Void and d2 /= Void
46   local
47     dice_result : INTEGER
48   do
49     d1.roll
50     print ("%Nd1:" + d1.face_value.out)
51     d2.roll
```

```

53     print ("%Nd2:" + d2.face_value.out)
        dice_result := d1.face_value + d2.face_value
        move (dice_result)
55     ensure
        player_moved: old location /= location
57     end

59     move (n: INTEGER)
        -- Move 'Current' n steps forward.
61     require
        n_consistent: n >= 2 and n <= 12
63     local
        i: INTEGER
65     do
        from
67         i := 1
        until
69         location = Void or else i > n
        loop
71         location := location.next
            i := i + 1
73     end
        end

75     invariant
77     name_exists: name /= Void and then not name.is_empty
79     end
    end

```

Listing 8: Class *BOARD*

```

class
2   BOARD

4   create
    make

6   feature -- Creation

8   make

10  -- Create a board with squares.
    local
12     i: INTEGER
        square_x, square_y: SQUARE
14     do
        from
16         i := 1
            create start_square.make ("Square" + i.out)
18         square_x := start_square
            i := i + 1
20     until
        i > max_number_of_squares

```

```
22   loop
23       create square_y.make ("Square" + i.out)
24       square_x.set_next (square_y)
25       square_x := square_y
26       i := i + 1
27   end
28 end

30 feature -- Access

32 max_number_of_squares: INTEGER = 40
33     -- The max number of squares supported by the current board.
34
35 start_square: SQUARE
36     -- The start square.

38 invariant
39     max_number_of_squares_consistent: max_number_of_squares = 40
40     start_square_exists : start_square /= Void

42 end
```

Listing 9: Class *SQUARE*

```
class
2  SQUARE

4  create
5      make

6
7  feature {NONE} -- Initialization
8
9      make (n: STRING)
10         -- Initialization for 'Current'.
11         require
12             name_exists: n /= Void and then not n.is_empty
13         do
14             name := n
15         ensure
16             name_set: name = n
17         end

18  feature -- Access

19
20     name: STRING
21         -- The square name.

22     next: SQUARE
23         -- The next square.

24  feature -- Status setting

25
26     set_next (sq: SQUARE)
```

```
30  -- Set next square.
    require
32    square_exists: sq /= Void
    do
34    next := sq
    ensure
36    next_square_set: next = sq
    end
38
    invariant
40  name_exists: name /= Void and then not name.is_empty
42 end
```