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*

```
indexing
  description: "Loopings class (Assignment 6)"

class
  LOOPINGS

inherit

  TOURISM

feature -- Explore Paris

  equip
      -- Build trams and connecting lines.
    do
      Paris.display
      wait
      from
        Paris.lines.start
      until
        Paris.lines.after
      loop
        generate_trams_for_line (Paris.lines.item_for_iteration)
        Paris.lines.forth
      end
      generate_connecting_bus_line (3, station_balard, station_mairie_d_issy)
    end

  generate_connecting_bus_line (n: INTEGER; start_station, end_station:
      TRAFFIC_STATION)
      -- Generate 'n' new stations and a bus line.
    require
      stations_exist: start_station /= Void and end_station /= Void
      stations_not_same: start_station /= end_station
      n_positive: n > 0
    local
      l: TRAFFIC_LINE
      s: TRAFFIC_STATION
      t: TRAFFIC_TYPE_BUS
      i: INTEGER
      v: TRAFFIC_POINT
    do
      v := (end_station.location - start_station.location)/(n+1)
      create t.make
      create l.make_with_terminal ("Bus line", t, start_station)
      Paris.put_line (l)
```

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

59
    generate_trams_for_line (a_line: TRAFFIC_LINE)
61      -- Generate trams for 'a_line' on every second station if allowed.
    require
63      a_line_exists: a_line /= Void
    local
65      t: TRAFFIC_TRAM
        type: TRAFFIC_TYPE_TRAM
67  do
        create type.make
69      if a_line.type.is_equal (type) then
          from
71          a_line.start
          until
73          a_line.after
          loop
75          create t.make_with_line (a_line)
            t.set_to_station (a_line.item)
77          t.start
            Paris.put_tram (t)
79          a_line.forth
            if not a_line.after then
81            a_line.forth
            end
83        end
      end
85  end

87 end
```

3 Loop painting

Solution

Listing 4: Class *DRAWING_MANAGER*

```
1 class
    DRAWING_MANAGER
3 create
```

```
  make
feature -- Initialization

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

      io.put_string ("%NCheckered triangle:%N%N")
      print_checker_triangle (n)

      io.put_new_line
      io.put_new_line

      io.put_string ("Checkered diamond:%N%N")
      print_checker_diamond (n)

    end

feature -- Checkerboards

  print_checker_triangle (n: INTEGER)
      -- Print a checker pyramid of size 'n' by 'n'.
    local
      i, j, star: INTEGER
    do
      from
        i := 1
        star := 0
      until
        i > n
      loop
        from
          j := 1
        until
          j > i
        loop
          if j \\ 2 = star then
            io.put_character (' ')
          else
            io.put_character ('*')
          end
          j := j + 1
        end
        star := 1 - star
        i := i + 1
        io.put_new_line
      end
```

```
    end

  print_checker_diamond (n: INTEGER)
      -- Print checker diamond of size 'n'.
    local
      i: INTEGER
      left, middle: STRING
    do
      create left.make_filled (' ', n)
      middle := ""
      from
        i := 1
      until
        i > n
      loop
        left.remove_tail (1)
        middle.append ("* ")
        io.put_string (left + middle + "%N")
        i := i + 1
      end
      from
        i := 1
      until
        i > n
      loop
        left.append (" ")
        middle.remove_tail (2)
        io.put_string (left + middle + "%N")
        i := i + 1
      end
    end
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 psudo-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
  create
5   make

7 feature {NONE} -- Initialization

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

31 feature -- Basic operations

33  play
        -- Start a game.
35    local
        i: INTEGER
37    do
```

```
        print ("%N*** Simple Boardgame ***")
39      from

41      until
          has_winner
43      loop
          from
45          i := 1
          until
47          has_winner or else i > number_of_players
          loop
49          players [i].play (die_1, die_2)
            if players [i].location = Void then
51            print ("%NAnd the winner is: " + players [i].name)
              has_winner := True
53          end
            i := i + 1
55        end
        end
57      print ("%N*** Game Over ***")
      ensure
59      game_has_winner: has_winner
      end
61
  feature -- Status
63
    game_board: BOARD
65      -- The game board.

67  number_of_players: INTEGER = 2
        -- For testing purposes, the number of players is set to 2.
69
    players: ARRAY [PLAYER]
71      -- Container for players.

73  die_1: DIE
        -- The first die.
75
    die_2: DIE
77      -- The second die.

79  has_winner: BOOLEAN
        -- Does the game have a winner?
81
  invariant
83
    game_board_exists: game_board /= Void
85
     players_exist : players /= Void and then not players.is_empty
87
    number_of_players_consistent: number_of_players >= 2 and number_of_players <= 6
89
```

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

Listing 6: Class *DIE*

```
class
  DIE

create
  make


feature {NONE} -- Initialization

  make
      -- Create a die with valid initial face value.
    do
      face_value := rand.item \\ 6 + 1
    end

feature -- Access

  face_value: INTEGER

feature -- Basic operations

  roll
      -- Roll die
  do
    rand.forth
    face_value := rand.item \\ 6 + 1
  end

feature {NONE} -- Implementation

  rand: RANDOM
      -- Pseudo-random number generator.
    local
      t: TIME
      seed: INTEGER
    once
      create t.make_now
      seed := (t.fine_seconds * 1000).rounded
      create Result.set_seed (seed)
      Result.start
    end


invariant
  six_sided_die : face_value >=1 and face_value <= 6

end
```

Listing 7: Class *PLAYER*

```
1 class
    PLAYER
3
  create
5   make


7
  feature -- Access
9
    name: STRING
11        -- Player name.

13    location: SQUARE
          -- Player current location.
15
  feature {NONE} -- Initialization
17
    make (n: STRING)
19        -- Create a player with name.
      require
21        name_exists: n /= Void and then not n.is_empty
      do
23        name := n
      ensure
25        name_set: name = n
      end
27
  feature -- Status setting
29
     set_location (loc: SQUARE)
31        -- Set location for player.
      require
33         location_exists : loc /= Void
      do
35        location := loc
      ensure
37         location_set : location = loc
      end
39
  feature -- Basic operations
41
    play (d1,d2: DIE)
43        -- Play a turn.
      require
45         dice_exist : d1 /= Void and d2 /= Void
      local
47         dice_result : INTEGER
      do
49        d1.roll
          print ("%Nd1:" + d1.face_value.out)
51        d2.roll
```

```
        print ("%Nd2:" + d2.face_value.out)
53      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
```

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
            create square_y.make ("Square" + i.out)
24          square_x.set_next (square_y)
            square_x := square_y
26          i := i + 1
          end
28      end

30  feature -- Access

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

38  invariant
      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
      make
6
  feature {NONE} -- Initialization
8
      make (n: STRING)
10        -- Initialization for 'Current'.
        require
12        name_exists: n /= Void and then not n.is_empty
        do
14        name := n
        ensure
16        name_set: name = n
        end
18
  feature -- Access
20
      name: STRING
22        -- The square name.

24    next: SQUARE
          -- The next square.
26
  feature -- Status setting
28
      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
```