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

```
expose 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

```
expose 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 class
5 LOOPINGS

7 inherit
9 TOURISM

11 feature -- Explore Paris

13 equip
   -- Build trams and connecting lines.
15 do
17 wait
19 from
21 until
23 loop
25 generate_trams_for_line (Paris.lines.item_for_iteration )
27 Paris.lines.forth
29 end
31 generate_connecting_bus_line (3, station_balard, station_mairie_d_issy )
33 end

29 generate_connecting_bus_line (n: INTEGER; start_station, end_station: TRAFFIC_STATION)
   -- Generate 'n' new stations and a bus line.

31 require
33 stations_exist : start_station /= Void and end_station /= Void
35 stations_not_same: start_station /= end_station
37 n_positive : n > 0

39 local
41 l: TRAFFIC_LINE
43 s: TRAFFIC_STATION
45 t: TRAFFIC_TYPE_BUS
47 i: INTEGER
49 v: TRAFFIC_POINT

41 do
43 v := (end_station.location - start_station.location)/(n+1)
45 create t.make
47 create l.make_with_terminal ("Bus line", t, start_station)
49 Paris.put_line (l)
from
i := 1
until
i > n
loop
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)
l.extend (s)
i := i + 1
end
l.extend (end_station)
end

generate_trams_for_line (a_line: TRAFFIC_LINE)
    -- Generate trams for 'a_line' on every second station if allowed.
require
    a_line_exists : a_line /= Void
local
t: TRAFFIC_TRAM
type: TRAFFIC_TYPE_TRAM
do
create type.make
if a_line.type.is_equal (type) then
    from
    a_line.start
    until
    a_line.after
loop
create t.make_with_line (a_line)
t.set_to_station (a_line.item)
t.start
Paris.put_tram (t)
a_line.forth
if not a_line.after then
    a_line.forth
end
end
end
end

3 Loop painting

Solution

Listing 4: Class DRAWING_MANAGER

class
    DRAWING_MANAGER
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
            end
            io.put_new_line
        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
   until
   i > n
   loop
      left.append (" ")
      middle.remove_tail (2)
      io.put_string (left + middle + "%N")
      i := i + 1
   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 pseudo-random sequences. This should be likely to result in rolling a lot of doubles (test it yourself).

Listing 5: Class GAME

```plaintext
1 class
2  GAME
3  create
4  make

7 feature {NONE} -- Initialization

9  make
   -- Run application.
11  local
12   i: INTEGER
13   p: PLAYER
14  do
15       create game_board.make
16       create die_1.make
17       create die_2.make
18       create players.make (1, number_of_players)
19     from
20       i := 1
21     until
22       i > players.count
23     loop
24       create p.make ("Player" + i.out)
25       p. set_location (game_board.start_square)
26       players [i] := p
27       i := i + 1
28   end
29  end

31 feature -- Basic operations

33  play
   -- Start a game.
35  local
36   i: INTEGER
37  do
```
```plaintext
print ("%N*** Simple Boardgame ***")
from

until
  has_winner
loop
  from
  i := 1
  until
    has_winner or else i > number_of_players
  loop
    players[i].play(die_1, die_2)
    if players[i].location = Void then
      print("%NAnd the winner is: " + players[i].name)
      has_winner := True
    end
  i := i + 1
end
end
print ("%N*** Game Over ***")
ensure
  game.has_winner: has_winner
end

feature -- Status

  game_board: BOARD
  -- The game board.

  number_of_players: INTEGER = 2
  -- For testing purposes, the number of players is set to 2.

  players: ARRAY [PLAYER]
  -- Container for players.

  die_1: DIE
  -- The first die.

  die_2: DIE
  -- The second die.

  has_winner: BOOLEAN
  -- Does the game have a winner?

invariant

  game_board_exists: game_board /= Void

  players_exist: players /= Void and then not players.is_empty

  number_of_players_consistent: number_of_players >= 2 and number_of_players <= 6
```
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

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

end
Listing 7: Class PLAYER

```java
class PLAYER

create

make

feature -- Access

name: STRING
  -- Player name.

location: SQUARE
  -- Player current location.

feature {NONE} -- Initialization

make (n: STRING)
  -- Create a player with name.
  require
    name_exists: n /= Void and then not n.is_empty
  do
    name := n
  ensure
    name_set: name = n
end

feature -- Status setting

set_location (loc: SQUARE)
  -- Set location for player.
  require
    location_exists: loc /= Void
  do
    location := loc
  ensure
    location_set: location = loc
end

feature -- Basic operations

play (d1,d2: DIE)
  -- Play a turn.
  require
    dice_exist: d1 /= Void and d2 /= Void
  local
    dice_result: INTEGER
  do
    d1.roll
    print ("%Nd1:" + d1.face_value.out)
    d2.roll
```

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

dice_result := d1.face_value + d2.face_value
move (dice_result)

ensure
playerMoved: old location /= location
end

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

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

Listing 8: Class BOARD

class 2 BOARD
create
    make
feature -- Creation
    make
        -- Create a board with squares.
        local
        do
            i := 1
            create start_square.make ("Square" + i.out)
        until
        i > max_number_of_squares
    square_x, square_y: SQUARE
end

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

feature -- Access

max_number_of_squares: INTEGER = 40
  -- The max number of squares supported by the current board.

start_square: SQUARE
  -- The start square.

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

end

Listing 9: Class SQUARE

class SQUARE

create
make

feature {NONE} -- Initialization

make (n: STRING)
  -- Initialization for 'Current'.
  require
    name_exists: n /= Void and then not n.is_empty
  do
    name := n
  ensure
    name_set: name = n
end

feature -- Access

name: STRING
  -- The square name.

next: SQUARE
  -- The next square.

feature -- Status setting

set_next (sq: SQUARE)
30      -- Set next square.
            require
            square_exists: sq /= Void
            do
            next := sq
            ensure
            next_square_set: next = sq
            end

38      invariant
40      name_exists: name /= Void and then not name.is_empty

42 end