Solution 6: Loops and conditionals

ETH Zurich

1 Reading loops

Version A:

- The result of the comparison using = will always be False (STRING is a reference type).
- The if-statement is inside the loop: it will highlight all the stations until it finds the right one.
- The corrected code of version A is shown in Listing 1.

Version B:

- Infinite 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: Version A

```pascal
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
   -- or: if found then
   Paris.stations.item_for_iteration.highlight
end
```

Listing 2: Version B

```pascal
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
```

2 Equipping Paris

Listing 3: Class LOOPINGS

```pascal
indexing
description: "Loopings class (Assignment 6)"
class LOOPINGS
inherit
   TOURISM
feature
   -- Explore Paris
equip
   -- Build trams and connecting lines.
do
   Paris.display
   wait
from
```

2
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 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
ensure
added_if_tram_line: a_line.type.is_equal (create { TRAFFIC_TYPE_TRAM }, make)
implies
Paris.trams.count = old Paris.trams.count + (a_line.count + 1) // 2
unchanged_if_not_tram_line: not a_line.type.is_equal (create { TRAFFIC_TYPE_TRAM }, make) implies
Paris.trams.count = old Paris.trams.count
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

\[
do
\]

\[
v := (\text{end} \_\text{station} \_\text{location} - \text{start} \_\text{station} \_\text{location}) / (n + 1)
\]

create t.make
create l.make_with_terminal ("Bus line", t, start_station)
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)
ensure
one \_more \_line: Paris.lines.count = old Paris.lines.count + 1
end

3 Loop painting

Listing 4: Class LOOP_PAINTING

class

LOOP_PAINTING
create
make

feature -- Initialization
make

local
n: INTEGER
do
io.put_string ("Enter a positive integer: ")
io.read_integer
n := io.last_integer

if n <= 0 then
print ("Wrong input")
else
io.put_string ("%NCheckered triangle:%N%N")
print_checker_triangle (n)

io.put_new_line
io.put_new_line
io.put_string("Checkered diamond:%d\n\n")
print_checker_diamond(n)
end
end

feature -- Painting

print_checker_triangle (n: INTEGER)
-- Print a checker triangle of size ‘n’.
require
positive_n: n > 0
local
i, j, space: INTEGER
do
from
i := 1
space := 0
until
i > n
loop
from
j := 1
until
j > i
loop
if j \ 2 = space then
io.put_character(' ')
else
io.put_character('*')
end
j := j + 1
end
space := 1 - space
i := i + 1
io.put_new_line
end
end

print_checker_diamond (n: INTEGER)
-- Print checker diamond of size ‘n’.
require
positive_n: n > 0
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

4 Boardgame: Part 2

Listing 5: Class GAME

class
  GAME

create
  make

feature {NONE} -- Initialization

  make (n: INTEGER)
  -- Create a game with 'n' players.
  require
    n_in_bounds: Min_player_count <= n and n <= Max_player_count
  local
    i: INTEGER
    p: PLAYER
  do
    create die_1.roll
    create die_2.roll
    create players.make (1, n)
  from
    i := 1
  until
    i > players.count
  loop
    create p.make ("Player" + i.out)
p.set_position (1)
players [i] := p
i := i + 1
end
end

feature -- Basic operations

play -- Start a game.
local
i: INTEGER
do
from
until
winner /= Void
loop
from
i := 1
until
winner /= Void or else i > players.count
loop
players [i].play (die_1, die_2)
if players [i].position > Square_count then
winner := players [i]
end
i := i + 1
end
end
ensure
has_winner: winner /= Void
end

feature -- Constants
Min_player_count: INTEGER = 2
-- Minimum number of players.

Max_player_count: INTEGER = 6
-- Maximum number of players.

Square_count: INTEGER = 40
-- Number of squares.

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

die_1: DIE
-- The first die.

die_2: DIE
-- The second die.
winner: \texttt{PLAYER} 
\hspace{1em} -- The winner (Void if the game if not over yet).

\textbf{invariant}

\hspace{1em} \texttt{dice\_exist: \texttt{die\_1} /\!\!= \texttt{Void} \textbf{and} \texttt{die\_2} /\!\!= \texttt{Void}}

\hspace{1em} \texttt{players\_exist: \texttt{players} /\!\!= \texttt{Void}}

\hspace{1em} \texttt{number\_of\_players\_consistent: Min\_player\_count \leq \texttt{players\_count} \textbf{and} \texttt{players\_count} \leq Max\_player\_count}

\textbf{class} \texttt{DIE}

\texttt{create roll}

\textbf{feature} \hspace{1em} -- Access

\hspace{1em} \texttt{Face\_count: INTEGER = 6} 
\hspace{1em} \hspace{1em} -- Number of faces.

\hspace{1em} \texttt{face\_value: INTEGER} 
\hspace{1em} \hspace{1em} -- Latest value.

\textbf{feature} \hspace{1em} -- Basic operations

\texttt{roll} 
\hspace{1em} \hspace{1em} -- Roll die.
\hspace{1em} \hspace{1em} \texttt{do}
\hspace{1em} \hspace{2em} \texttt{random\_forth}
\hspace{1em} \hspace{2em} \hspace{1em} \texttt{face\_value := random\_item \ \\} Face\_count + 1}
\hspace{1em} \texttt{end}

\textbf{feature} \{\texttt{NONE}\} \hspace{1em} -- Implementation

\hspace{1em} \texttt{random: RANDOM} 
\hspace{1em} \hspace{1em} -- Random sequence.
\hspace{1em} \hspace{1em} \texttt{local}
\hspace{1em} \hspace{2em} \texttt{t: TIME}
\hspace{1em} \hspace{1em} \texttt{once}
\hspace{1em} \hspace{2em} \hspace{1em} \texttt{create t\_make\_now}
\hspace{1em} \hspace{2em} \hspace{2em} \texttt{create Result\_set\_seed (t\_milli\_second)}
\hspace{1em} \hspace{2em} \hspace{1em} \texttt{Result\_start}
\hspace{1em} \texttt{end}

\textbf{invariant}
class PLAYER

create
make

feature {NONE} -- Initialization
make (n: STRING)
   -- Create a player with name ‘n’.
   require
   name_exists: n /= Void and then not n.is_empty
   do
   name := n.twin
   ensure
   name_set: name ~ n
   end

feature -- Access

name: STRING
   -- Player name.

position: INTEGER
   -- Current position on the board.

feature -- Moving

set_position (pos: INTEGER)
   -- Set position to ‘pos’.
   do
   position := pos
   ensure
   position_set: position = pos
   end

feature -- Basic operations

play (d1, d2: DIE)
   -- Play a turn with dice ‘d1’, ‘d2’.
   require
   dice_exist: d1 /= Void and d2 /= Void
   do
   d1.roll
   d2.roll
   set_position (position + d1.face_value + d2.face_value)

Listing 7: Class PLAYER

face_value_valid: face_value >= 1 and face_value <= Face_count
end
print (name + ” rolled ” + d1.face_value.out + ” and ” + d2.face_value.out + ”.
Moves to ” + position.out + ”.%N”)
end

invariant

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

Listing 8: Class APPLICATION

class APPLICATION

create
make

feature

make
-- Launch the application.
local
count: INTEGER
game: GAME
do
print (”%N*** Simple Boardgame ***%N”)
from
count := {GAME}.Min_player_count - 1
until
{GAME}.Min_player_count <= count and count <= {GAME}.Max_player_count
loop
print (”Enter number of players between ” + {GAME}.Min_player_count.out + ” and ” + {GAME}.Max_player_count.out + ”: ”)
io.read_integer
count := io.last_integer
end
create game.make (count)
game.play
print (”%NAnd the winner is: ” + game.winner.name)
print (”%N*** Game Over ***”)
end
end