Solution 6: Loopy games

ETH Zurich

1 Loop painting

Listing 1: Class LOOP_PAINTING

note
description: "Drawing figures with asterisks."

class LOOP_PAINTING

create
make

feature -- Initialization

make
-- Get size and paint.
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
print("%NCheckered triangle:%N%N")
print_checker_triangle (n)

print("%N%N")

print("Checkered diamond:%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 \(\div\) 2 = space then
    print (' ')
  else
    print ('*')
  end
  j := j + 1
end
space := 1 − space
i := i + 1
print ("%N")
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 ("* ")
  print (left + middle + "%N")
  i := i + 1
end
from
  i := 1
until
  i > n
loop
left.append (" ")
middle.remove_tail (2)
print (left + middle + "]%N")
i := i + 1
end
end
end

2 Bagels

Listing 2: Class BAGELS

note
description : ”Bagels application”

class
BAGELS

create
execute, set_answer

feature -- Initialization
execute
  -- Play bagels.
local
d: INTEGER
do
  io.put_string (”*** Welcome to Bagels! ***%N”)
from
  io.last_integer > 0
until
  io.read_integer
loop
  io.put_string (”Enter the number of digits (positive):%N”)
  end
  d := io.last_integer
  play (d)
end

feature -- Implementation

play (d: INTEGER)
  -- Generate a number with ‘d’ digits and let the player guess it.
require
d_positive: d > 0
local
guess_count: INTEGER
  guess: STRING
do
  io.put_string (”I’m thinking of a number...”)
generate_answer (d)
io.put_string (” Okay, got it!%N”)
from
until
    guess ~ answer
loop
    io.put_string ("Enter your guess: ")
    io.read_line
    guess := io.last_string
    if guess.count = d and guess.is_natural and not guess.has ('0') then
        print (clue (guess) + "%N")
        guess_count := guess_count + 1
    else
        io.put_string ("Incorrect input: please enter a positive number with " + d.
                        out + " digits containing no zeros%N")
    end
end
print ("Congratulations! You made it in " + guess_count.out + " guesses.")
end

answer: STRING
    -- Correct answer.
set_answer (s: STRING)
    -- Set 'answer' to 's'.
require
    s.non_empty: s /= Void and then not s.is_empty
    is_natural: s.is_natural
    no_zeros: not s.has ('0')
do
    answer := s
ensure
    answer_set:answer = s
end

generate_answer (d: INTEGER)
    -- Generate a number with 'd' nonzero digits and store it in 'answer'.
require
    d.positive: d > 0
local
    random: V.Random
    i: INTEGER
do
    create answer.make_filled (' ', d)
    create random
    from
        i := 1
    until
        i > d
    loop
        answer[i] := (random.bounded_item (1, 9)).out [1]
        random.forth
        i := i + 1
    end
end
ensure
  answer_exists: answer /= Void
  correct_length: answer.count = d
  is_natural: answer.is_natural
  no_zeros: not answer.has ('0')
end

clue (guess: STRING): STRING
-- Clue for 'guess' with respect to 'answer'.
require
  answer_exists: answer /= Void
  guess_exists: guess /= Void
  same_length: answer.count = guess.count
local
  i, k: INTEGER
  answer_copy, guess_copy: STRING
do
  Result := ""
  answer_copy := answer.twin
  guess_copy := guess.twin
  from
    i := 1
  until
    i > answer_copy.count
  loop
    if answer_copy [i] = guess_copy [i] then
      Result := Result + "Fermi ",
      answer_copy [i] := ' ',
      guess_copy [i] := ' ',
    end
    i := i + 1
  end
  from
    i := 1
  until
    i > answer_copy.count
  loop
    if answer_copy [i] /= ' ' then
      k := guess_copy.index_of (answer_copy [i], 1)
      if k > 0 then
        Result := Result + "Pico ",
        guess_copy [k] := ' ',
      end
      i := i + 1
    end
  end
  if Result.is_empty then
    Result := "Bagels"
  end
ensure
  result_exists: Result /= Void
3 Board game: Part 2

Listing 3: 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
  print (p.name + " joined the game.%N")
  i := i + 1
end
print ("%N")

feature -- Basic operations

play
  -- Start a game.
local
  round, i: INTEGER
do
  from
    round := 1
  print ("The game begins.%N")
  print_board
until
winner /= Void
loop
  print ("%NRound #" + round.out + "%N")
  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
  print_board
  round := round + 1
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: V.ARRAY [PLAYER]
  -- Container for players.

  die_1: DIE
  -- The first die.

  die_2: DIE
  -- The second die.

  winner: PLAYER
  -- The winner (Void if the game if not over yet).

feature {NONE} -- Implementation

  print_board
  -- Output players positions on the board.
local
i, j: INTEGER
board: STRING

do
  io.new_line
  board := "."
  board.multiply (Square_count)
  print (board)
  io.new_line
from
  i := 1
until
  i > players.count
loop
  from
    j := 1
  until
    j >= players [i].position
loop
  print (" ")
  j := j + 1
end
  print (i)
  io.new_line
  i := i + 1
end

invariant
dice_exist: die_1 /= Void and die_2 /= Void
players_exist: players /= Void
number_of_players_consistent: Min_player_count <= players.count and players.count <= Max_player_count
end

Listing 4: Class DIE

class
  DIE
create
  roll

feature -- Access
  Face_count: INTEGER = 6
    -- Number of faces.

  face_value: INTEGER
    -- Latest value.

feature -- Basic operations
  roll
    -- Roll die.
do
  random.forth
  face_value := random.bounded_item (1, Face_count)
end

feature {NONE} -- Implementation

random: V_RANDOM
  -- Random sequence.
  once
  create Result
end

invariant
  face_value_valid: face_value >= 1 and face_value <= Face_count
end

Listing 5: Class PLAYER

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
feature — Basic operations

\( \text{play} \ (d1, \ d2: \text{DIE}) \)

\(--\) Play a turn with dice ‘d1’, ‘d2’.

\textbf{require}
\( \text{dice\_exist: } \ d1 \neq \text{Void} \ \text{and} \ \ d2 \neq \text{Void} \)
\textbf{do}
\( \ d1.\text{roll} \)
\( \ d2.\text{roll} \)
\( \ \text{set\_position} \ (\text{position} + \ d1.\text{face\_value} \ + \ d2.\text{face\_value}) \)
\( \ \text{print} \ ( \text{name} + \ " \ \text{rolled} \ \" + \ d1.\text{face\_value}.out \ + \ " \ \text{and} \ \" + \ d2.\text{face\_value}.out + \ "). \)
\( \ \text{Moves to} \ " + \ \text{position}.out + \ "\%N") \)
\textbf{end}

\textbf{invariant}
\( \text{name\_exists: } \ \text{name} \neq \text{Void} \ \text{and} \ \text{then not name.is\_empty} \)
\textbf{end}

Listing 6: Class \textit{APPLICATION}

class \textit{APPLICATION}

create

\textit{make}

\textbf{feature}

\textit{make}

\(--\) Launch the application.

\textbf{local}
\( \ \text{count} : \text{INTEGER} \)
\( \ \text{game} : \text{GAME} \)
\textbf{do}
\( \ \text{from} \)
\( \ \text{count} := \{\text{GAME}\}.\text{Min\_player\_count} - 1 \)
\( \ \textbf{until} \)
\( \ \{\text{GAME}\}.\text{Min\_player\_count} \leq \ \text{count} \ \text{and} \ \text{count} \leq \ {\text{GAME}\}.\text{Max\_player\_count} \)
\textbf{loop}
\( \ \text{print\ ("Enter number of players between }" + \{\text{GAME}\}.\text{Min\_player\_count}.out + \ "\ \text{and} \ " + \{\text{GAME}\}.\text{Max\_player\_count}.out + \ ": \") \)
\( \ \text{io.read\_integer} \)
\( \ \text{count} := \ \text{io.last\_integer} \)
\textbf{end}

\textbf{create} \ \text{game.make (count)}
\( \ \text{game}\_\text{play} \)
\( \ \text{print\ ("And the winner is: }" + \ \text{game}\_\text{winner}.\text{name}) \)
\( \ \text{print\ ("\%N\star\star\star \text{Game Over } \star\star\star")} \)
\textbf{end}
end
4 MOOC: programming exercises

Listing 7: Class DECIMAL_TO_BINARY_CONVERTER

note
description: "Summary description for {DECIMAL_TO_BINARY_CONVERTER
                }."
author: ""
date: "$Date$"
revision: "$Revision$

class
    DECIMAL_TO_BINARY_CONVERTER

feature -- Conversion

valid_input (n: INTEGER): BOOLEAN
    -- Is 'n' a valid input for a conversion?
do
    Result := 0 <= n and n <= 100000000
end

to_binary (n: INTEGER): STRING
    -- Binary representation of a number 'n' expressed in base 10.
require
    valid_input: valid_input (n)
local
    my_local: INTEGER
do
    if n = 0 then
        Result := "0"
    else
        from
            -- We will build the result string digit by digit
            Result := ""
            -- We start from n and save it in our temp variable
            my_local := n
        invariant
            -- Our invariant states that at every iteration the value
            -- in our temp variable corresponds to n divided by 2
            -- to the power of the number of elements in the result string.
            -- The truncation to an integer is necessary because ^ gives a real.
            my_local = n \ (2 ^ Result.count).truncated_to_integer
        until
            -- We exit the loop when my_local reaches 0
            my_local = 0
        loop
            -- We build the result string one digit at the time
            -- Note that we are using the modulus operator
            -- for computing the remainder of integer division
            Result.prepend_integer (my_local \ 2)
            -- Now we update my_local using the integer division
my_local := my_local // 2

variant
    -- This is always decreasing and positive
    my_local + 1
end
end

ensure
    result_exists: result /= Void and not Result.is_empty
end
end

Listing 8: Class WORD_GAMES

note
description: "The class {PALINDROME} implements algorithms that are related
to strings."
author: "hce"
date: "11.07.2013"

class
    WORD_GAMES

feature -- Basic algorithms

is_palindrome (s: STRING): BOOLEAN
    -- Returns true if 's' is a palindrome.
require
    input_valid: s /= Void and not s.is_empty
local
    l_reversed_s: STRING
    i: INTEGER
do
    -- We start with an empty reversed string.
    l_reversed_s := ""
    Result := false
from
    i := s.count
until
    i = 0
loop
    -- We append to the reversed string the characters from
    -- the s, read from the end to the beginning (in reverse order)
    l_reversed_s.append_character (s.at (i))
    i := i − 1
end
    -- If a string is the same as its reversed, then it is palindrome.
if l_reversed_s.is_equal (s) then
    Result := true
end
end