Solution 8: Recursion

ETH Zurich

1 An infectious task

1. Correct. This version works and uses tail recursion. It will always give the flu to \( p \) first, and then call \( \text{infect} \) on his/her coworker. The recursion ends when either there is no coworker, or the coworker is already infected. Without the second condition the recursion is endless if the coworker structure is cyclic.

2. Incorrect. This version results in endless recursion if the coworker structure is cyclic. The main cause is that the coworker does not get infected before the recursive call is made, so with a cyclic structure nobody will ever be infected to terminate the recursion.

3. Incorrect. This version results in an endless loop if the structure is cyclic. The main problem is with the loop’s exit condition that does not include the case when \( q \) is already infected.

4. Correct. However, this version will call \( \text{set\_flu} \) twice on all reachable persons except the initial one. On the initial person \( \text{set\_flu} \) will be called once in case of a non-circular structure and three times in case of a circular structure.

Multiple coworkers

```plaintext
class
    PERSON

create
    make

feature  --  Initialization

make (a_name: STRING)
    --  Create a person named ‘a_name’.
    require
        a_name_valid: a_name /= Void and then not a_name.is_empty
    do
        name := a_name
        create { V.ARRAYED_LIST[PERSON] } coworkers
    ensure
        name_set: name = a_name
        no_coworkers: coworkers.is_empty
    end

feature  --  Access
```
name: STRING
    -- Name.

coworkers: V_LIST [PERSON]
    -- List of coworkers.

has_flu: BOOLEAN
    -- Does the person have flu?

feature -- Element change

add_coworker (p: PERSON)
    -- Add 'p' to 'coworkers'.
          require
                p_exists: p /= Void
                p_different: p /= Current
                not_has_p: not coworkers.has (p)
          do
                coworkers.extend_back (p)
          ensure
                coworker_set: coworkers.has (p)
          end

set_flu
    -- Set 'has_flu' to True.
          do
                has_flu := True
          ensure
                has_flu: has_flu
          end

invariant
    name_valid: name /= Void and then not name.is_empty
    coworkers_exists: coworkers /= Void
end

infect (p: PERSON)
    -- Infect 'p' and coworkers.
          require
                p_exists: p /= Void
          do
                p.set_flu
                across
                p.coworkers as c
                loop
                    if not c.item.has_flu then
                        infect (c.item)
                    end
                end
          end
end

The coworkers structure is a directed graph. The master solution traverses this graph using
depth-first search.

2 Short trips

Listing 1: Class SHORT_TRIPS

note
description: "Short trips."

class SHORT_TRIPS

inherit ZURICH_OBJECTS

feature -- Explore Zurich

highlight_short_distance (s: STATION)
  -- Highlight stations reachable from ‘s’ within 2 minutes.
  require
  station_exists: s /= Void
  do
    highlight_reachable (s, 2 * 60)
  end

feature {NONE} -- Implementation

highlight_reachable (s: STATION; t: REAL_64)
  -- Highlight stations reachable from ‘s’ within ‘t’ seconds.
  require
    station_exists: s /= Void
  local
    line: LINE
    next: STATION
  do
    if t >= 0.0 then
      Zurich_map.station_view (s).highlight
      across
      s.lines as li
      loop
        line := li.item
        next := line.next_station (s, line.north_terminal)
        if next /= Void then
          highlight_reachable (next, t - s.position.distance (next.position) / line.speed)
        end
        next := line.next_station (s, line.south_terminal)
        if next /= Void then
          highlight_reachable (next, t - s.position.distance (next.position) / line.speed)
        end
      end
    end
end
end
3 Get me out of this maze!

Listing 2: Class MAZE

class MAZE

inherit ARRAY2[CHARACTER]
  redefine out
end

create make

feature -- Map characters

Empty_char: CHARACTER = '·'
  -- Character for empty fields.

Exit_char: CHARACTER = '★'
  -- Character for an exit field.

Wall_char: CHARACTER = '#'
  -- Character for a wall field.

Visited_char: CHARACTER = 'x'
  -- Character for a field that has been visited by 'find_path'.

feature -- Element change

set_empty (r, c: INTEGER)
  -- Set field with row ‘r’ and column ‘c’ to empty.
  require
    r_valid: r >= 1 and r <= height
    c_valid: c >= 1 and c <= width
  do
    put (Empty_char, r, c)
  ensure
    field_set: item (r, c) = Empty_char
end

set_exit (r, c: INTEGER)
  -- Set field with row ‘r’ and column ‘c’ to exit.
  require
    r_valid: r >= 1 and r <= height
    c_valid: c >= 1 and c <= width
  do
    put (Exit_char, r, c)
ensure
    field_set: item (r, c) = Exit_char
end

set_wall (r, c: INTEGER)
    -- Set field with row ‘r’ and column ‘c’ to wall.
require
    r_valid: r >= 1 and r <= height
    c_valid: c >= 1 and c <= width
do
    put (Wall_char, r, c)
ensure
    field_set: item (r, c) = Wall_char
end

set_visited (r, c: INTEGER)
    -- Set field with row ‘r’ and column ‘c’ to visited.
require
    r_valid: r >= 1 and r <= height
    c_valid: c >= 1 and c <= width
do
    put (Visited_char, r, c)
ensure
    field_set: item (r, c) = Visited_char
end

feature -- Status report

is_valid (c: CHARACTER): BOOLEAN
    -- Is ‘c’ a valid map character?
do
    Result := c = Empty_char or c = Wall_char or c = Exit_char
end

feature -- Path finding

path: STRING
    -- Sequence of instructions to find the way out of the maze.

find_path (r, c: INTEGER)
    -- Find the path starting at row ‘r’ and column ‘c’.
require
    row_valid: 1 <= r and r <= height
    column_valid: 1 <= c and c <= width
do
    if item (r, c) = Exit_char then
        path := ""
    elseif item (r, c) = Empty_char then
        set_visited (r, c)
        if (c - 1) > 0 and path = Void then
            find_path (r, c - 1)
        if path /= Void then
            path := path + "L"
        end
    elseif item (r, c) = Wall_char then
        if (c + 1) <= width then
            find_path (r, c + 1)
        end
    end
end
path := "W > " + path
end
end
if (r - 1) > 0 and path = Void then
    find_path (r - 1, c)
    if path /= Void then
        path := "N > " + path
    end
end
end
if (c + 1) <= width and path = Void then
    find_path (r, c + 1)
    if path /= Void then
        path := "E > " + path
    end
end
end
if (r + 1) <= height and path = Void then
    find_path (r + 1, c)
    if path /= Void then
        path := "S > " + path
    end
end
set_empty (r, c)
end
end

feature -- Output

out: STRING
    -- Maze map.
local
    i, j: INTEGER
    do
        from
        i := 1
        j := 1
        Result := ""
        until
        i > height
        loop
            from
            j := 1
            until
            j > width
            loop
                Result.append_character (item (i, j))
                j := j + 1
            end
            i := i + 1
        end
        Result := Result + "\%N"
    end
Listing 3: Class \textit{APPLICATION}

\begin{verbatim}
class MAZE_APPLICATION
create
make

feature -- Initialization
make
    -- Run application.
    local
        mr: MAZE_READER
        maze: MAZE
        start_row, start_column: INTEGER
    do
        create mr
        Io.put_string ("Please enter the name of a maze file: ")
        Io.read_line
        mr.read_maze (Io.last_string)
        if mr.has_error then
            Io.put_string (mr.error_message)
        else
            maze := mr.last_maze
            Io.put_string ("%N" + maze.out + "%N")
            Io.put_string ("Please enter a starting field for finding a path.\n")
            from until start_row /= 0
            loop
                Io.put_string ("Row: ")
                Io.read_integer
                if Io.last_integer > 0 and Io.last_integer <= maze.height then
                    start_row := Io.last_integer
                else
                    Io.put_string ("Invalid row. Please try again\n")
                end
            end
            from until start_column /= 0
            loop
                Io.put_string ("Column: ")
                Io.read_integer
                if Io.last_integer > 0 and Io.last_integer <= maze.width then
                    start_column := Io.last_integer
                else
                    Io.put_string ("Invalid column. Please try again\n")
                end
            end
        end
    end
end
\end{verbatim}
maze.find_path (start_row, start_column)
if maze.path /= Void then
    Io.put_string ("There’s a way out! Go ” + maze.path.out + ”You’re free!%N”)
else
    Io.put_string ("Oops, no way out! You’re trapped!%N")
end
end
end

end -- class APPLICATION