Solution 9: Recursion

ETH Zurich

1 An infectious task

1. Correct. This version works and uses tail recursion. It will always give flu to p first, and then call *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 be ever 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 *set_flu* twice on all reachable persons except the initial one. On the initial person *set_flu* will be called once in case of non-circular structure and three times in case of circular structure.

Multiple coworkers

```
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 coworkers.make
    ensure
      name_set: name = a_name
      coworkers_exists: coworkers /= Void
    end

feature -- Access
  name: STRING
```

```
  coworkers: LINKED_LIST [PERSON]

  has_flu: BOOLEAN

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 (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
      from
        p.coworkers.start
      until
        p.coworkers.off
      loop
        if not p.coworkers.item.has_flu then
          infect (p.coworkers.item)
        end
        p.coworkers.forth
      end
    end
```

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

2 Reachable stations

Listing 1: Class *RECURSIVE_HIGHLIGHTING*

```
note
  description: "Recursive highlighting class (Assignment 9)"
  date: "$Date$"
  revision: "$Revision$"

class
  RECURSIVE_HIGHLIGHTING

inherit
  TOURISM

feature -- Explore Paris
  show
      -- Highlight stations that are reachable within a certain time limit.
    do
      Paris.display
      highlight_reachable_stations (Station_chatelet, 10.0)
    end

  highlight_reachable_stations (s: TRAFFIC_STATION; t: REAL_64)
      -- Highlight all stations that are reachable from 's' within travel time 't'.
    require
      s_exists: s /= Void
      t_positive: t > 0.0
    local
      stop: TRAFFIC_STOP
      i: INTEGER
    do
      s.highlight
      from
        i := 1
      until
        i > s.stops.count
      loop
        stop := s.stops.i_th (i)
        if stop.right /= Void and then (t - stop.time_to_next) >= 0.0 then
          highlight_reachable_stations (stop.right.station, t - stop.time_to_next)
        end
        i := i + 1
      end
    end

end
```

3 Get me out of this maze!

Listing 2: Class *MAZE_READER*

```
class
  MAZE_READER
```

```
feature -- Basic operations.
  read_maze (f: STRING)
      -- Read a maze from file with filename 'f'.
    local
      file: PLAIN_TEXT_FILE
      n, m, i: INTEGER
    do
      has_error := False
      error_message := ""
      create file.make (f)
      if not file.exists then
        has_error := True
        error_message := "File " + f.out + " does not exist.%N"
      else
        file.open_read
        if not file.is_open_read then
          has_error := True
          error_message := "File " + f.out + " could not be opened.%N"
        else
          file.start
          file.read_integer
          n := file.last_integer
          file.read_integer
          m := file.last_integer
          if n <= 0 or m <= 0 then
            has_error := True
            error_message := "Maze dimensions not valid.%N"
          else
            from
              i := 0
              create last_maze.make (m, n)
            until
              file.off or has_error or i >= n*m
            loop
              file.read_character
              inspect file.last_character
              when {MAZE}.empty_char then
                last_maze.set_empty ((i // n) + 1, (i \\ n) + 1)
              when {MAZE}.wall_char then
                last_maze.set_wall ((i // n) + 1, (i \\ n) + 1)
              when {MAZE}.exit_char then
                last_maze.set_exit ((i // n) + 1, (i \\ n) + 1)
              else
                if file.last_character.is_space then
                  -- Ignore it
                  i := i - 1
                else
                  has_error := True
                  error_message := "Wrong character " + file.last_character.out + "%N"
                end
              end
              i := i + 1
```

```
            end
            if i < n * m then
              has_error := True
              error_message := "Maze not filled%N"
            end
          end
        end
      end
    end

feature -- Access
  has_error: BOOLEAN
      -- Has there been an error when reading?

  error_message: STRING
      -- Error message.

  last_maze: MAZE
      -- Maze that was read last.
end
```

Listing 3: Class *MAZE*

```
class
  MAZE

inherit
  ARRAY2 [CHARACTER]
    redefine
      out
    end

create
  make

feature -- Constants
  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 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 out of the maze.

  path_exists: BOOLEAN
```

```
      -- Does a path exist?

  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_exists := True
        path := ""
      elseif item (r, c) = Empty_char then
        set_visited (r, c)
        if (c - 1) > 0 and not path_exists then
          find_path (r, c - 1)
          if path_exists then
            path := "W > " + path
          end
        end
        if (r - 1) > 0 and not path_exists then
          find_path (r - 1, c)
          if path_exists then
            path := "N > " + path
          end
        end
        if (c + 1) <= width and not path_exists then
          find_path (r, c + 1)
          if path_exists then
            path := "E > " + path
          end
        end
        if (r + 1) <= height and not path_exists then
          find_path (r + 1, c)
          if path_exists then
            path := "S > " + path
          end
        end
        set_empty (r, c)
      end
    ensure
      path_exists_consistent: path_exists = (path /= Void)
    end

feature -- Output
  out: STRING
      -- Output
    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
        Result := Result + "%N"
      end
    end

end
```

Listing 4: Class *APPLICATION*

```
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
        maze.find_path (start_row, start_column)
        if maze.path_exists 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
```