Solution 8: Recursion

ETH Zurich

1 An infectious task

1. 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 a non-circular structure and three times in case of a circular structure.

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. This version works and uses tail recursion. It will always give the 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.

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 {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
   all_coworkers_exist: not coworkers.has (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
The coworkers structure is a directed graph. The master solution traverses this graph using \textit{depth-first search}.

## 2 Short trips

Listing 1: Class \textit{SHORT\_TRIPS}

\begin{verbatim}
\textbf{note} \\
\textit{description}: "Short trips."

\textbf{class} \textit{SHORT\_TRIPS}

\textbf{inherit} \textit{ZURICH\_OBJECTS}

\textbf{feature} \textit{-- Explore Zurich}

\textit{highlight\_short\_distance} (s: \textit{STATION}) \\
\textit{-- Highlight stations reachable from ‘s’ within 2 minutes.}

\textbf{require} \\
\textit{station\_exists: s /= Void}

\textbf{do} \\
\textit{highlight\_reachable} (s, 2 * 60)

\textbf{end}

\textbf{feature} \{\textit{NONE}\} \textit{-- Implementation}

\textit{highlight\_reachable} (s: \textit{STATION}; t: \textit{REAL\_64}) \\
\textit{-- Highlight stations reachable from ‘s’ within ‘t’ seconds.}

\textbf{require} \\
\textit{station\_exists: s /= Void}

\textbf{local} \\
\textit{line: LINE}

\textit{next: STATION}

\textbf{do} \\
\textbf{if} \textit{t} \textbf{>=} 0.0 \textbf{then} \\
\hspace{1em} \textit{Zurich\_map\_view} (s).\textit{highlight}

\hspace{1em} \textbf{across}

\hspace{1em} \textit{s\_lines as li}

\hspace{1em} \textbf{loop}

\hspace{2em} \textit{line} := \textit{li\_item}

\hspace{2em} \textit{next} := \textit{line\_next\_station} (s, \textit{line\_north\_terminal})

\hspace{2em} \textbf{if} \textit{next} \textbf{=} \textit{Void} \textbf{then}

\hspace{3em} \textit{highlight\_reachable} (\textit{next}, \textit{t} \textbf{-} \textit{s\_position\_distance} \textit{(next\_position)} \textbf{/} \textit{line\_speed})

\hspace{2em} \textbf{end}

\hspace{2em} \textit{next} := \textit{line\_next\_station} (s, \textit{line\_south\_terminal})

\hspace{2em} \textbf{if} \textit{next} \textbf{=} \textit{Void} \textbf{then}

\hspace{3em} \textit{highlight\_reachable} (\textit{next}, \textit{t} \textbf{-} \textit{s\_position\_distance} \textit{(next\_position)} \textbf{/} \textit{line\_speed})

\hspace{2em} \textbf{end}

\hspace{1em} \textbf{end}

\hspace{1em} \textbf{end}

\hspace{1em} \textbf{end}
\end{verbatim}
3 N Queens

Listing 2: Class PUZZLE

note
description: "N-queens puzzle."

class PUZZLE

feature -- Access

  size: INTEGER
  -- Size of the board.

  solutions: LIST [SOLUTION]
  -- All solutions found by the last call to 'solve'.

feature -- Basic operations

  solve (n: INTEGER)
  -- Solve the puzzle for 'n' queens
  -- and store all solutions in 'solutions'.
  require
  n_positive: n > 0
  do
    size := n
    create {LINKED_LIST [SOLUTION]} solutions.make
    complete (create {SOLUTION}.make_empty)
  ensure
    solutions_exists: solutions /= Void
    complete_solutions: across solutions as s all s.item.row_count = n end
  end

feature {NONE} -- Implementation

  complete (partial: SOLUTION)
  -- Find all complete solutions that extend the partial solution 'partial'
  -- and add them to 'solutions'.
  require
    partial_exists: partial /= Void
  local
    c: INTEGER
  do
    if partial.row_count = size then
      solutions.extend (partial)
    else
      from
        c := 1
until
  \( c > \text{size} \)
loop
  if not under_attack (partial, c) then
    complete (partial.extended_with (c))
  end
  \( c := c + 1 \)
end
end

\textbf{under\_attack (partial: SOLUTION; c: INTEGER): BOOLEAN}

\begin{itemize}
\item Is column ‘c’ of the current row under attack
\item by any queen already placed in partial solution ‘partial’?
\end{itemize}

\textbf{require}

\begin{itemize}
\item partial_exists: partial /\neq \text{Void}
\item column_positive: c \geq 0
\end{itemize}

\textbf{local}

\begin{itemize}
\item current_row, row: INTEGER
\end{itemize}

\textbf{do}

\begin{itemize}
\item current_row := partial.row_count + 1
\item from
\item \( row := 1 \)
\item until
\item Result or row > partial.row_count
\item loop
\item Result := attack_each_other (row, partial.column_at (row), current_row, c)
\item row := row + 1
\item end
\item end
\end{itemize}

\textbf{attack_each_other (row1, col1, row2, col2: INTEGER): BOOLEAN}

\begin{itemize}
\item Do queens in positions (‘row1’, ‘col1’) and (‘row2’, ‘col2’) attack each other?
\end{itemize}

\textbf{do}

\begin{itemize}
\item Result := row1 = row2 or
\item col1 = col2 or
\item (row1 - row2).abs = (col1 - col2).abs
\item end
\end{itemize}

\textbf{end}