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 *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)
      -- Highight 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)
      -- Highight 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 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 > size
        loop
```

```
          if not under_attack (partial, c) then
            complete (partial.extended_with (c))
          end
          c := c + 1
        end
      end
    end

  under_attack (partial: SOLUTION; c: INTEGER): BOOLEAN
      -- Is column 'c' of the current row under attack
      -- by any queen already placed in partial solution 'partial'?
    require
      partial_exists: partial /= Void
      column_positive: c > 0
    local
      current_row, row: INTEGER
    do
      current_row := partial.row_count + 1
      from
        row := 1
      until
        Result or row > partial.row_count
      loop
        Result := attack_each_other (row, partial.column_at (row), current_row, c)
        row := row + 1
      end
    end

  attack_each_other (row1, col1, row2, col2: INTEGER): BOOLEAN
      -- Do queens in positions ('row1', 'col1') and ('row2', 'col2') attack each other?
    do
      Result := row1 = row2 or
        col1 = col2 or
        (row1 - row2).abs = (col1 - col2).abs
    end

end
```

4 MOOC: Design by Contract, recursion

The order in which the questions and the answers appear here in the solution may vary because they are randomly shuffled at each attempt.

Design by Contract: preconditions

- In class KNIGHT you have feature *set_reputation* (*rep*: *INTEGER*). What precondition would you write for it? *rep* >= −5 **and** *rep* <= 5
- In class KNIGHT you have feature *attack_monster* (*mon*: *MONSTER*; *wep*: *WEAPON*). What precondition would you write for it? *wep* /= **Void and** *mon* /= **Void and then** *wep.is_ready*

- In class MONSTER you have feature *scan_direction* (*dir*: *DIRECTION*). What precondition would you write for it? No explicit precondition is needed.
- n class WEAPON you have feature *set_ready* (*wep_ready*: *BOOLEAN*). What precondition would you write for it? No precondition is needed here.
- Suppose that in class MONSTER, feature attack, you want to add the expression *is_knight_close* to the existing precondition *is_angry*. The true sentence is: The compound precondition *is_angry* **and** *is_knight_close* is a stronger precondition than *is_angry*.
- Suppose you know that a knight can only fight in battle if his or her hit points are greater than 10. Which is a reasonable precondition for BOOLEAN feature *is_fit_for_battle* in class KNIGHT? No precondition is needed here.

Design by Contract: postconditions

- In class KNIGHT you have feature *set_reputation* (*rep*: *INTEGER*). What postcondition would you write for it? *reputation* = *rep*
- In class KNIGHT you have feature *attack_monster* (*mon*: *MONSTER*; *wep*: *WEAPON*). What postcondition would you write for it? **old** *mon.hit_points* >= *mon.hit_points* **and not** *wep.is_ready*
- In class MONSTER you have feature *scan_direction* (*dir*: *DIRECTION*). What postcondition would you write for it? *is_knight_found* **or** *is_scanning_complete*.
- In class WEAPON you have feature *set_ready* (*wep_ready*: *BOOLEAN*). What postcondition would you write for it? *is_ready* = *wep_ready*.
- Suppose that in class KNIGHT, feature attack, you want to add to the existing postcondition *old_mon.hit_points* >= *mon.hit_points* **and not** *wep_is_ready* the new clause: *reputation* = **old** *reputation* + 1 **or** *reputation* = 5. The true sentence is: The compound postcondition: *old_mon.hit_points* >= *mon.hit_points* **and not** *wep_is_ready* **and** (*reputation* = **old** *reputation* + 1 **or** *reputation* = 5) is a stronger postcondition than the pre-existing postcondition.
- Suppose you know that a knight can only fight in battle if his or her hit points are greater than 10. Which is a reasonable postcondition for BOOLEAN feature *is_fit_for_battle* in class KNIGHT? **Result implies** *hit_points* > 10.

Design by Contract: class invariants

- Given what you know about class KNIGHT, what invariant would you write? *reputation* >= −5 **and** *reputation* <= 5 **and** *hit_points* >= 0
- Given what you know about class MONSTER, what invariant would you write? *hit_points* >= 0
- Given what you know about class WEAPON, what invariant would you write? *is_magic* **implies** *is_ready* **and** *damage* >= 1.
- Given what you know about class DIRECTION, what invariant would you write? *internal_direction* = 1 **or** *internal_direction* = 2 **or** *internal_direction* = 3 **or** *internal_direction* = 4.

Design by Contract: contracts and inheritance

- Given what you know about class KNIGHT_MAGE, which precondition clause would you write for feature *attack_monster* (*mon*: *MONSTER*; *wep*: *WEAPON*)? **require else** *mana* > 0
- Given what you know about class KNIGHT_MAGE, which postcondition clause would you write for feature *attack_monster* (*mon*: *MONSTER*; *wep*: *WEAPON*)? **ensure then** *mana* < **old** *mana*
- Given what you know about class KNIGHT_MAGE, which class invariant would you write for it? *mana* >= 0.
- Given what you know about class GOBLIN, which precondition would you write for feature *attack_with_weapon* (*kni*: *KNIGHT*; *wep*: *WEAPON*)? **require** *last_knight_found* = *kni* **and** *is_angry* **and** *wep.is_ready* .
- Given what you know about class GOBLIN, which postcondition would you write for feature *attack_with_weapon* (*kni*: *KNIGHT*; *wep*: *WEAPON*)? **ensure** *is_angry*.
- Given what you know about class GOBLIN, which class invariant would you write for it? No invariant clause is needed.

Design by Contract: putting it all together

- Assume a class FILTER receiving input data from a class INPUT_HANDLER that in turn is used to validate user input. The following statements are true: To check for user input correctness, you should not be using preconditions in class INPUT_HANDLER, but use if statements instead; To check for user input correctness, you should be using preconditions in class FILTER instead of if statements.
- Assume that the correct precondition for a feature *f* (*s*: *STRING*) is: *pre*: *s* /= **Void and then** *s* = **"test"** Consider now the following precondition: *pre2*: *s* /= **Void and then not** *s.is_empty* The following statements are true: pre2 is an over-approximation of pre; pre2 is complete and unsound.
- Assume that the correct precondition for a feature *f* (*s*: *STRING*) is: *pre*: *s* /= **Void and then not** *s.is_empty* Consider now the following precondition: *pre2*: *s* /= **Void and then** *s* = **"test"** The following statements are true: pre2 is an under-approximation of pre; pre2 is incomplete and sound.
- Assume that the correct postcondition for a feature f is: *post*: *s* /= **Void and then not** *s.is_empty* Where *s*: *STRING* is an attribute. Consider now the following postcondition: *post2*: *s* /= **Void and then** *s* = **"test"**. The following statements are true: post2 is an under-approximation of pre; post2 is too strong; post2 is sound but incomplete.
- Assume that the correct postcondition for a feature f is: *post*: *s* /= **Void and then** *s* = **"test"** Where *s*: *STRING* is an attribute. Consider now the following postcondition: *post2*: *s* /= **Void and then not** *s.is_empty* The following statements are true: post2 is an over-approximation of post; post2 is complete and unsound; post2 is too weak.

Recursion

- The correct way to complete the code of the routine *countdown* is the following:

```
countdown (n: INTEGER)
        -- Count down from n to 0.
    do
        if n >= 0 then
            print (n.out)
            countdown (n-1)
        else
            --nothing here
        end
    end
```

- The following routine, when called with n having value 4, keeps printing consecutive numbers starting from 4, and goes into an infinite loop:

```
countdown (n: INTEGER)
    do
        if n > 0 then
            print (n.out)
            countdown (n+1)
        else
            print ("Done")
        end
    end
```

- The following routine, when called with n having value 4, prints "4321Done":

```
countdown (n: INTEGER)
    do
        if n > 0 then
            print (n.out)
            countdown (n-1)
        else
            print ("Done")
        end
    end
```

- If a routine r calls another routine s, which calls another routine t, which finally calls routine s, then routine s is recursive (direct recursion) and routine t is recursive (direct recursion).

Programming exercise: recursive algorithm for gcd

Listing 3: Class *RECURSIVE_GCD*

```
note
  description: "Encapsulates a recursive algorithm for computing the gcd of two
      positive integers."
  author: "mp"
  date: "$Date$"
  revision: "$Revision$"

class
  RECURSIVE_GCD
```

```
feature -- Basic operations

  gcd (a, b: INTEGER): INTEGER
      -- Greater common divisor between a and b.
    require
      a_positive: a > 0
      b_positive: b > 0
    do
      -- This solution is from Dijkstra.
      -- It is based on the observation that if a > b,
      -- then gcd (a,b) = gcd (a-b,b)
        if a = b then
                Result := a
            else if a > b then
                  Result := gcd (a-b, b)
                 else
                    Result := gcd (a, b-a)
                 end
            end
    ensure
      result_positive: Result > 0
    end
end
```