1 Terminology (10 points)

Goal
This task will test your understanding of the object-oriented programming concepts presented so far in the lecture. This is a multiple-choice test.

Todo
Place a check-mark in the box if the statement is true. There may be multiple true statements per question; 0.5 points are awarded for checking a true statement or leaving a false statement un-checked, 0 points are awarded otherwise.

1. Objects and classes
   □ a. All types are either reference or expanded.
   □ b. If an object is of an expanded type, its fields cannot be modified at runtime.
   □ c. Suppliers of class $C$ can use all the features of class $C$.
   □ d. A class can be both a supplier and a client.
   □ e. If $C$ is a deferred class, then no entity can exist in a program with static type $C$.

2. About loops:
   □ a. A loop must always define an invariant, otherwise the program will not compile.
   □ b. The variant of the loop must increase with every loop iteration.
   □ c. The variant of the loop must decrease with every loop iteration and must always be $\geq 0$.
   □ d. It is possible that a loop will never terminate.

3. Information hiding ...
   □ a. ... is the technique of presenting client programmers with an interface that only contains the public features of a class.
b. ... is the technique of presenting client programmers with an interface that includes only features that have built-in security controls.

c. ... is the technique of presenting client programmers with an interface that includes a superset of the properties of a software element.

d. ... is the technique of presenting client programmers with an interface that includes only a subset of the properties of a software element.

4. Inheritance and polymorphism

a. A deferred class cannot inherit from an effective class.

b. A class \( C \) cannot inherit from two different classes \( A1 \) and \( A2 \), if both \( A1 \) and \( A2 \) have a common ancestor class.

c. An instruction \( o.f \) at runtime can result in executing different routines.

d. An entity of static type \( C \) can only be attached to an object of a type that is an ancestor of \( C \).

e. In class \( C \) a feature \( f \) inherited from class \( A \) can only be redefined if \( f \) is deferred in \( A \).

5. Design by Contract

a. The creation procedure only needs to ensure that the invariant of the created object holds at the end of the procedure body.

b. Every procedure ensures that the postcondition \( \textbf{True} \) holds.

c. The class invariant needs to hold before every procedure call.

d. A procedure \( pp \), that redefines another procedure \( p \), needs to ensure the postcondition of procedure \( p \).

e. A procedure \( pp \), that redefines another procedure \( p \), can provide a precondition that is stronger than the one given by procedure \( p \).

Solution

1. Objects and classes

a. All types are either reference or expanded.

b. If an object is of an expanded type, its fields cannot be modified at runtime.

c. Suppliers of class \( C \) can use all the features of class \( C \).

d. A class can be both a supplier and a client.

e. If \( C \) is a deferred class, then no entity can exist in a program with static type \( C \).

2. About loops:

a. A loop must always define an invariant, otherwise the program will not compile.

b. The variant of a loop must increase with every loop iteration.

c. The variant of a loop must decrease with every loop iteration and must always be \( \geq 0 \).

d. It is possible that a loop will never terminate.

3. Information hiding ...
a. ... is the technique of presenting client programmers with an interface that only contains the public features of a class.

b. ... is the technique of presenting client programmers with an interface that includes only features that have built-in security controls.

c. ... is the technique of presenting client programmers with an interface that includes a superset of the properties of a software element.

d. ... is the technique of presenting client programmers with an interface that includes only a subset of the properties of a software element.

✓

4. Inheritance and polymorphism

a. A deferred class cannot inherit from an effective class.

b. A class \( C \) cannot inherit from two different classes \( A1 \) and \( A2 \), if both \( A1 \) and \( A2 \) have a common ancestor class.

✓

c. An instruction \( o.f \) at runtime can result in executing different routines.

d. An entity of static type \( C \) can only be attached to an object of a type that is an ancestor of \( C \).

e. In class \( C \) a feature \( f \) inherited from class \( A \) can only be redefined if \( f \) is deferred in \( A \).

5. Design by Contract

a. The creation procedure only needs to ensure that the invariant of the created object holds at the end of the procedure body.

✓

b. Every procedure ensures that the postcondition \( \text{True} \) holds.

c. The class invariant needs to hold before every procedure call.

✓

d. A procedure \( pp \), that redefines another procedure \( p \), needs to ensure the postcondition of procedure \( p \).

e. A procedure \( pp \), that redefines another procedure \( p \), can provide a precondition that is stronger than the one given by procedure \( p \).

2 Design by Contract (10 Points)

Class \( \text{PERSON} \) is part of a software system that models marriage relations between persons. The following rules do not necessarily have universal value but describe a particular set of rules for marriage at a particular time and place in the past, e.g. Canton Zürich 1900:

1. Every person has a nonempty name.
2. A person cannot be married to himself/herself.
3. If a person \( X \) is married to a person \( Y \), then \( Y \) is married to \( X \).
4. In order for a person \( X \) to be able to marry a person \( Y \), neither \( X \) nor \( Y \) may be already married.
5. Divorces are not allowed.

Your task is to fill in the contracts of the class (preconditions, postconditions and class invariant) according to the specification given. You are not allowed to change the class interfaces or any of the already given implementations. Note that the number of dotted lines does not indicate the number of necessary code lines that you have to provide.
class PERSON
create make

feature {NONE} -- Creation
  make (n: STRING)
    -- Create a person with a name ‘n’.
    require
    do
      -- Create a copy of the argument and assign it to ‘name’
      name := n.twin
    ensure
  end

feature -- Access
  name: STRING
    -- Person’s name.
  spouse: PERSON
    -- Spouse if a spouse exists, Void otherwise.

feature -- Status report
  is_married: BOOLEAN
    -- Is person married?
    do
      Result := (spouse /= Void)
    ensure
end
feature \{ \texttt{PERSON} \} -- Implementation

\hspace{1em} \texttt{accept\_marriage} \ (p: \texttt{PERSON})
\hspace{1em} -- Set 'spouse' to 'p', who is already married to you.

\hspace{1em} \texttt{require}

\hspace{1em} \texttt{do}
\hspace{2em} \texttt{spouse} := p
\hspace{2em} \texttt{ensure}

\hspace{2em} \texttt{end}

feature -- Basic operations

\hspace{1em} \texttt{marry} \ (p: \texttt{PERSON})
\hspace{1em} -- Marry 'p'.

\hspace{1em} \texttt{require}

\hspace{1em} \texttt{do}
\hspace{2em} \texttt{spouse} := p
\hspace{2em} p.\texttt{accept\_marriage} \ (\texttt{Current})
\hspace{2em} \texttt{ensure}

\hspace{2em} \texttt{end}
Solution

class
  PERSON

create
  make

  feature {NONE} -- Creation
    make (n: STRING)
      -- Create a person with a name ‘n’.
      require
        n_nonempty: n /= Void and then not n.is_empty
      do
        -- Create a copy of the argument and assign it to name
        name := n.twin
      ensure
        name_set: n ~ name
        not_married yet: not is_married
      end

  feature -- Access
    name: STRING
      -- Person’s name.
    spouse: PERSON
      -- Spouse if a spouse exists, Void otherwise.

  feature -- Status report
is_married: BOOLEAN
   -- Is person married?
   do
      Result := (spouse /= Void)
   end

feature {PERSON} -- Implementation

accept_marriage (p: PERSON)
   -- Set 'spouse' to 'p', who is already married to you.
   require
      p_exists: p /= Void
      p_not_current: p /= Current
      current_not_married: not is_married
      target_maybe_married: p.spouse = Current
   do
      spouse := p
   ensure
      spouse_set: spouse = p
      is_married: is_married
   end

feature -- Basic operations

marry (p: PERSON)
   -- Marry 'p'.
   require
      p_exists: p /= Void
      p_not_current: p /= Current
      current_not_married: not is_married
      target_not_married: not p.is_married
   do
      spouse := p
      p.accept_marriage (Current)
   ensure
      current_spouse_is_p: spouse = p
   end

invariant
   name_nonempty: name /= Void and then not name.is_empty
   is_married_if_spouse_exists: is_married = (spouse /= Void)
   irreflexive_marriage: spouse /= Current
   symmetric_marriage: is_married implies (spouse.spouse = Current)

3 Digital root (10 points)

The digital root (Quersumme) of a number is found by adding together the digits that make up the number. If the resulting number has more than one digit, the process is repeated until a single digit remains.
Example input and output

<table>
<thead>
<tr>
<th>Input</th>
<th>Digital root</th>
<th>Computation process</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>6</td>
<td>= 1 + 2 + 3</td>
</tr>
<tr>
<td>5720</td>
<td>5</td>
<td>= 1 + 4 ← 14 = 5 + 7 + 2 + 0</td>
</tr>
<tr>
<td>99999999</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Your task in this problem is to implement a function that, given a non-negative number, calculates the digital root and returns it as the result. Fill in the body of function `digital_root` below. Your implementation should work with `INTEGER` objects only. You might find the following two operators of class `INTEGER` useful: \ (modulo) and // (integer division).

There exists a closed-form solution to this problem:

\[
\text{digital\_root}(n) = \begin{cases} 
0 & \text{if } n = 0, \\
9 & \text{if } n /\not\equiv 0 \text{ and then } n \mod 9 = 0, \\
9 - \left\lfloor \frac{n}{9} \right\rfloor & \text{otherwise.}
\end{cases}
\]

You are not allowed to use this to solve this programming exercise!

1 `digital_root (a\_number: INTEGER): INTEGER`  
   \hspace{1cm} \-- Digital root (Quersumme) of `a\_number`
3 `require`  
5 \hspace{1cm} `a\_number\_positive: a\_number >= 0`  
7 `local`  
9 `do`  
11 `local`  
13 `do`  
15 `do`  
17 `do`  
19 `do`  
21 `do`  
23 `do`  
25 `do`  
27 `do`  
29 `do`  
31 `do`  
33 `do`
ensure
  result_in_range : 0 <= Result and Result <= 9
end

Solution

digital_root  (a_number: INTEGER): INTEGER
  -- Digital root (Quersumme) of 'a_number'
  require
    a_number_within_range: a_number >= 0
  local
    number: INTEGER
  do
    from
      Result := a_number
    invariant
      result_non_negative : Result >= 0
    until
      Result < 10
  loop
    from
      number := Result
      Result := 0
    invariant
      -- 'Result' is a sum of i lower digits of 'old Result'
      -- 'number' contains n - i upper digits of 'old Result'
    until
      number = 0
  loop
    Result := Result + (number \ 10)
    number := number // 10
  variant
4 Doubly linked lists (14 points)

In the lecture you have been taught about singly linked lists, which enables list traversal in one direction. In this task you have to implement a data structure called a **doubly linked list**, which should allow traversal in both directions. The structure consists of two classes: `INTEGER_LIST_CELL` and `INTEGER_LIST`. An object of type `INTEGER_LIST_CELL` holds an `INTEGER` as the cell content and has a `previous` and a `next` reference to two other objects of type `INTEGER_LIST_CELL`. By attaching the previous and next references correctly, two or more cells can be connected to form a list. The class `INTEGER_LIST` offers functionality to access the first and the last cell of a list, to add a new cell at the end, and to look for a specific value in the list. In Figure 1 you see a drawing of a doubly linked list.

![Doubly linked list](image)

**Figure 1: Doubly linked list**

Read through the class `INTEGER_LIST_CELL` in Listing 4. You will need the features of this class for the rest of the task.

1. Implement the feature `extend` of class `INTEGER_LIST` (see Listing 3). This feature takes an `INTEGER` as argument, generates a new object of type `INTEGER_LIST_CELL` with the given `INTEGER` as content and puts the new cell at the end of the list. Make sure that your implementation satisfies the given postcondition of the feature.

2. Implement the feature `has` of class `INTEGER_LIST` (see Listing 3). This feature checks if the value it receives as argument is contained in any cell of the list. In the example of Figure 1, the first cell contains the value 18, the second cell contains the value 3, and the third one contains the value 12.
create
make_empty

feature -- Initialization
make_empty is
-- Initialize the list to be empty.
do
  first := Void
  last := Void
  count := 0
end

feature -- Access
first : INTEGER_LIST_CELL
-- Head element of the list, Void if the list is empty
last : INTEGER_LIST_CELL
-- Tail element of the list, Void if the list is empty

feature -- Measurement
count: INTEGER
-- Number of cells in the list

feature -- Element change
extend (a_value: INTEGER) is
-- Append an integer list cell with content 'a_value' at the end of the list.
local
  el: INTEGER_LIST_CELL
do
ensure

one more: \( \text{count} = \text{old count} + 1 \)

first_set : \( \text{count} = 1 \) implies first.value = a_value

last_set : last.value = a_value
end

feature -- Status report

empty: BOOLEAN is

-- Is the list empty?
do
  Result := (count = 0)
end

has (a_value: INTEGER): BOOLEAN is

-- Does the list contain a cell with value ‘a_value’?
local

...
Listing 2: Class \texttt{INTEGER\_LIST\_CELL}

class \texttt{INTEGER\_LIST\_CELL}

create

\texttt{set\_value}

feature \ -- Access

\texttt{value: \texttt{INTEGER}}

\texttt{\ -- Content that is stored in the list cell}

\texttt{next: \texttt{INTEGER\_LIST\_CELL}}

\texttt{\ -- Reference to the next integer list cell of a list}

\texttt{previous: \texttt{INTEGER\_LIST\_CELL}}

\texttt{\ -- Reference to the previous integer list cell of a list}

feature \ -- Element change

\texttt{set\_value \ (x: \texttt{INTEGER}) is}

\texttt{\ \ -- Set \textquote{value} to \textquote{x}.}

\texttt{do}

\texttt{value := x}

\texttt{ensure}

\texttt{value\_set: value = x}

\texttt{end}

\texttt{set\_next \ (el: \texttt{INTEGER\_LIST\_CELL}) is}

\texttt{\ -- Set \textquote{next} to \textquote{el}.}

\texttt{do}

\texttt{next := el}

\texttt{ensure}

\texttt{next\_set: next = el}

\texttt{end}

\texttt{set\_previous \ (el: \texttt{INTEGER\_LIST\_CELL}) is}

\texttt{\ -- Set \textquote{previous} to \textquote{el}.}

\texttt{do}

\texttt{previous := el}
39   ensure
40       previous_set: previous = el
41   end
43 end

Solution

Listing 3: Solution class INTEGER_LIST

class INTEGER_LIST
create
make_empty

feature -- Initialization
make_empty is
   -- Initialize the list to be empty.
do
   first := void
   last := void
   count := 0
end

feature -- Access
first: INTEGER_LIST_CELL
   -- Head element of the list, Void if the list is empty
last: INTEGER_LIST_CELL
   -- Tail element of the list, Void if the list is empty

feature -- Element change
extend (a_value: INTEGER) is
   -- Append a integer list cell with content ‘a_value’ at the end of the list.
local
   el: INTEGER_LIST_CELL
do
   create el, set_value (a_value)
   if empty then
      first := el
   else
      last, set_next (el)
      el, set_previous (last)
   end
   last := el
   count := count + 1
ensure
   one_more: count = old count + 1
first_set: count = 1 implies first.value = a_value
last_set: last.value = a_value

feature -- Measurement

count: INTEGER
    -- Number of cells in the list

feature -- Status report

has (a_value: INTEGER): BOOLEAN is
    -- Does the list contain a cell with value ‘a_value’?
local
cursor: INTEGER_LIST_CELL
do
    from cursor := first
    until cursor = Void or Result
    loop
        if cursor.value = a_value then
            Result := True
        end
    cursor := cursor.next
end

empty: BOOLEAN is
    -- Is the list empty?
do
    Result := (count = 0)
end

Listing 4: Class INTEGER_LIST_CELL

class INTEGER_LIST_CELL
create
    set_value

feature -- Access

value: INTEGER
    -- Content that is stored in the list cell

next: INTEGER_LIST_CELL
    -- Reference to the next integer list cell of a list

previous: INTEGER_LIST_CELL
    -- Reference to the previous integer list cell of a list
17 feature -- Element change

19 set_value (x: INTEGER) is
   -- Set 'value' to 'x'.
21   do
23     value := x
25   ensure
   value_set: value = x
27 end

27 set_next (el: INTEGER_LIST_CELL) is
   -- Set 'next' to 'el'.
29   do
31     next := el
33   ensure
   next_set: next = el
35 end

35 set_previous (el: INTEGER_LIST_CELL) is
   -- Set 'previous' to 'el'.
37   do
39     previous := el
41   ensure
   previous_set: previous = el
43 end