Mock Exam 1

ETH Zurich

November 7, 2012

Name: _____

Group: _____

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 ≥ 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 `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 ≥ 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 `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 *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

    ..........

    ..........
```

```
54
56          end

58 feature {PERSON} -- Implementation

60      accept_marriage (p: PERSON)
                -- Set 'spouse' to 'p', who is already married to you.
62          require

64

66

68

70
            do
72              spouse := p
            ensure
74

76

78

80

82          end

84 feature -- Basic operations

86      marry (p: PERSON)
                -- Marry 'p'.
88          require

90

92

94

96
            do
98              spouse := p
                p.accept_marriage (Current)
100         ensure
102

104
```

```
    .....
106
    .....
108         end

110 invariant

112 .....

114 .....

116 .....

118 .....

120 .....

122 .....
    end
```

Solution

```
    class
2       PERSON

4   create
        make
6
    feature {NONE} -- Creation
8
        make (n: STRING)
10              -- Create a person with a name 'n'.
            require
12              n_nonempty: n /= Void and then not n.is_empty
            do
14           -- Create a copy of the argument and assign it to name
                name := n.twin
16          ensure
                name_set: n ~ name
18              not_married_yet: not is_married
            end
20
    feature -- Access
22
        name: STRING
24          -- Person's name.

26      spouse: PERSON
            -- Spouse if a spouse exists, Void otherwise.
28
    feature -- Status report
```

```
30
    is_married: BOOLEAN
32          -- Is person married?
        do
34          Result := (spouse /= Void)
        end
36
feature {PERSON} -- Implementation
38
    accept_marriage (p: PERSON)
40          -- Set 'spouse' to 'p', who is already married to you.
        require
42          p_exists: p /= Void
            p_not_current: p /= Current
44          current_not_married: not is_married
            target_maybe_married: p.spouse = Current
46      do
            spouse := p
48      ensure
            spouse_set: spouse = p
50          is_married: is_married
        end
52
feature -- Basic operations
54
    marry (p: PERSON)
56          -- Marry 'p'.
        require
58          p_exists: p /= Void
            p_not_current: p /= Current
60          current_not_married: not is_married
            target_not_married: not p.is_married
62      do
            spouse := p
64          p.accept_marriage (Current)
        ensure
66          current_spouse_is_p: spouse = p
        end
68
invariant
70  name_nonempty: name /= Void and then not name.is_empty
     is_married_if_spouse_exists : is_married = (spouse /= Void)
72   irreflexive_marriage : spouse /= Current
    symmetric_marriage: is_married implies (spouse.spouse = Current)
74
end
```

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

Input	Digital root	Computation process
123	6	$= 1 + 2 + 3$
5720	5	$= 1 + 4 \leftarrow 14 = 5 + 7 + 2 + 0$
99999999	9	
8	8	

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:

$$digital\_root(n) = \begin{cases} 0 & \text{if } n = 0, \\ 9 & \text{if } n /= 0 \text{ and then } n \setminus\setminus 9 = 0, \\ n - 9\lfloor \frac{n}{9} \rfloor & \text{otherwise.} \end{cases}$$

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

```
1   digital_root  (a_number: INTEGER): INTEGER
      -- Digital root (Quersumme) of 'a_number'
3   require
      a_number_positive: a_number >= 0
5   local

7     .....

9     .....

11    .....
    do
13
      .....
15
      .....
17
      .....
19
      .....
21
      .....
23
      .....
25
      .....
27
      .....
29
      .....
31
      .....
33
      .....
```

```
35
      ..........
37
      ..........
39
      ..........
41
      ..........
43
      ..........
45
      ..........
47
      ..........
49
      ..........
51
      ..........
53
  ensure
55    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
```

```
        number
      end
    variant
      Result
    end
  end
```

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.

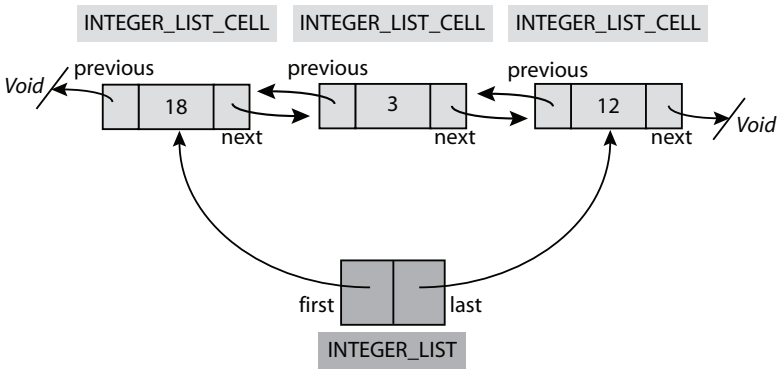

Figure 1: Doubly linked list

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.

Listing 1: Class *INTEGER_LIST*

```
class INTEGER_LIST
2
```

```
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
            .....

            .....

            .....

            .....

            .....

            .....

            .....

            .....

            .....

            .....
```

```
50        ...
52        ...
54        ...
56        ...
58        ...
60        ...
      ensure
62      one_more: count = old count + 1
          first_set : count = 1 implies first.value = a_value
64        last_set : last.value = a_value
      end
66
  feature -- Status report
68    empty: BOOLEAN is
        -- Is the list empty?
70    do
        Result := (count = 0)
72    end

74    has (a_value: INTEGER): BOOLEAN is
          -- Does the list contain a cell with value 'a_value'?
76    local
        ...
78
        ...
80
        ...
82    do
        ...
84
        ...
86
        ...
88
        ...
90
        ...
92
        ...
94
        ...
96
        ...
98
        ...
100
        ...
```

```
102
        .....
104
        .....
106
        .....
108
        .....
110
        .....
112     end
    end
```

Listing 2: Class *INTEGER_LIST_CELL*

```
1  class INTEGER_LIST_CELL

3  create
       set_value
5
   feature -- Access
7
       value: INTEGER
9          -- Content that is stored in the list cell

11     next: INTEGER_LIST_CELL
           -- Reference to the next integer list cell of a list
13
       previous: INTEGER_LIST_CELL
15         -- 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
           value := x
23     ensure
           value_set: value = x
25     end

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

35     set_previous (el: INTEGER_LIST_CELL) is
           -- Set 'previous' to 'el'.
37     do
           previous := el
```

```
39      ensure
          previous_set: previous = el
41      end

43  end
```

Solution

Listing 3: Solution class *INTEGER_LIST*

```
1  class
     INTEGER_LIST
3
   create
5    make_empty

7  feature -- Initialization

9    make_empty is
         -- Initialize the list to be empty.
11     do
         first := void
13       last := void
         count := 0
15     end

17 feature -- Access

19   first : INTEGER_LIST_CELL
         -- Head element of the list, Void if the list is empty
21
     last: INTEGER_LIST_CELL
23       -- Tail element of the list , Void if the list is empty

25 feature -- Element change

27   extend (a_value: INTEGER) is
         -- Append a integer list cell with content 'a_value' at the end of the list .
29     local
         el: INTEGER_LIST_CELL
31     do
         create el.set_value (a_value)
33       if empty then
           first := el
35       else
           last.set_next (el)
37         el.set_previous (last)
         end
39       last := el
         count := count + 1
41     ensure
         one_more: count = old count + 1
```

```
43        first_set : count = 1 implies first.value = a_value
          last_set : last.value = a_value
45    end

47 feature -- Measurement

49  count: INTEGER
        -- Number of cells in the list
51
  feature -- Status report
53
    has (a_value: INTEGER): BOOLEAN is
55      -- Does the list contain a cell with value 'a_value'?
      local
57      cursor: INTEGER_LIST_CELL
      do
59      from
          cursor := first
61      until
          cursor = Void or Result
63      loop
          if cursor.value = a_value then
65          Result := True
          end
67        cursor := cursor.next
        end
69    end

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

77 end
```

Listing 4: Class *INTEGER_LIST_CELL*

```
1 class INTEGER_LIST_CELL

3 create
    set_value
5
  feature -- Access
7
    value: INTEGER
9       -- Content that is stored in the list cell

11  next: INTEGER_LIST_CELL
        -- Reference to the next integer list cell of a list
13
    previous: INTEGER_LIST_CELL
15      -- 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
       value := x
23   ensure
       value_set: value = x
25   end

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

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

43 end
```