

Mock exam 1

ETH Zurich

Date: 9./10. November 2009

Name: _____

Group: _____

1 Terminology (8 points)

Put checkmarks in the checkboxes corresponding to the correct answers. Multiple correct answers are possible; there is at least one correct answer per question. A correctly set checkmark is worth 1 point, an incorrectly set checkmark is worth -1 point. If the sum of your points is negative, you will receive 0 points.

Example:

Which of the following statements are true?

1.
 - a. Classes exist only in the software text; objects exist only during the execution of the software.
 - b. Each object is an instance of its generic class.
 - c. An object is deferred if it has at least one deferred feature.

Which of the following statements are true?

1. A command...
 - a. is a query that is not implemented as an attribute.
 - b. may modify an object.
 - c. may appear in the precondition and the postcondition of another command but not in the precondition or the postcondition of a query.
 - d. may appear in the class invariant.
2. A query...
 - a. may be used as a creation procedure.
 - b. may be implemented as a routine.
 - c. may appear in the precondition and the postcondition of another query but not in the precondition or the postcondition of a command.
 - d. may appear in the class invariant.

3. A class...
 - a. is the description of a set of possible run-time objects to which the same features are applicable.
 - b. can only exist at runtime.
 - c. cannot be declared as expanded; only objects can be expanded.
 - d. may have more than one creation procedure.
4. Immediately before a successful execution of a creation instruction with target x of type C ...
 - a. $x = Void$ must hold.
 - b. $x \neq Void$ must hold.
 - c. the postcondition of the creation procedure may not hold.
 - d. the precondition of the creation procedure may not hold.
5. Immediately after a successful execution of a creation instruction with target x of type C ...
 - a. $x = Void$ must hold.
 - b. the postcondition of the creation procedure may not hold.
 - c. the precondition of the creation procedure may not hold.
 - d. the object attached to x satisfies the invariant of C .

2 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
123	6
5720	5
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).

Listing 1: Feature `digital_root`

```

2  digital_root (a_number: INTEGER): INTEGER
3  -- Digital root (Quersumme) of 'a_number'
4  require
5  a_number_positive: a_number >= 0 and a_number <= a_number.max_value
6  local
7
8  .....
9  .....
10 .....
    
```

```
12  do
14  .....
16  .....
18  .....
20  .....
22  .....
24  .....
26  .....
28  .....
30  .....
32  .....
34  .....
36  .....
38  .....
40  .....
42  .....
44  .....
46  .....
48  .....
50  .....
52  .....
54  .....
56  .....
58  .....
60  .....
62  ensure
    result_in_range : 0 <= Result and Result <= 9
end
```

3 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. A person cannot be married to himself/herself.
2. If a person X is married to a person Y, then Y is married to X.
3. In order for a person X to be able to marry a person Y, neither X nor Y may be already married.

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
2 create make

4 feature -- Access

6   name: STRING
      -- Person's name

8
10  spouse: PERSON
      -- Spouse if a spouse exists, Void otherwise

12 feature -- Creation

14   make (n: STRING)
      -- Create a person with a name
16   require

18 .....
20 .....
22 .....
24 .....
26   do
      -- Create a copy of the argument and assign it to 'name'
      name := n.twin
28   ensure

30 .....
32 .....
34 .....
36 .....

end
```

```
38 feature -- Status report
40     is_married: BOOLEAN
42         -- Is current person married?
43     do
44         Result := (spouse /= Void)
45     ensure
46
47 .....
48 .....
49 .....
50 .....
51 .....
52 .....
53 .....
54     end
55
56 feature {PERSON} -- Implementation
57     set_spouse (p: PERSON)
58         -- Set spouse to p
59     require
60
61 .....
62 .....
63 .....
64 .....
65 .....
66 .....
67 .....
68     do
69         spouse := p
70     ensure
71
72 .....
73 .....
74 .....
75 .....
76 .....
77 .....
78     end
79
80 feature -- Basic operations
81
82     marry (p: PERSON)
83         -- Get married to p
84     require
85
86 .....
87 .....
88 .....
```

```
90 .....  
92 .....  
94     do  
       set_spouse (p)  
       p.set_spouse (Current)  
     ensure  
98 .....  
100 .....  
102 .....  
104 .....  
106     end  
107 invariant  
108 .....  
110 .....  
112 .....  
114 .....  
116 end
```

4 Doubly linked lists (14 points)

In the lecture you have been taught about singly linked lists, which allow to move through the list in one direction. In this task you have to implement a data structure called a *doubly linked list*, which should allow moving in both directions through the list. 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.

Read through the class `INTEGER_LIST_CELL` in Listing 3. 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 2). 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 2). 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.

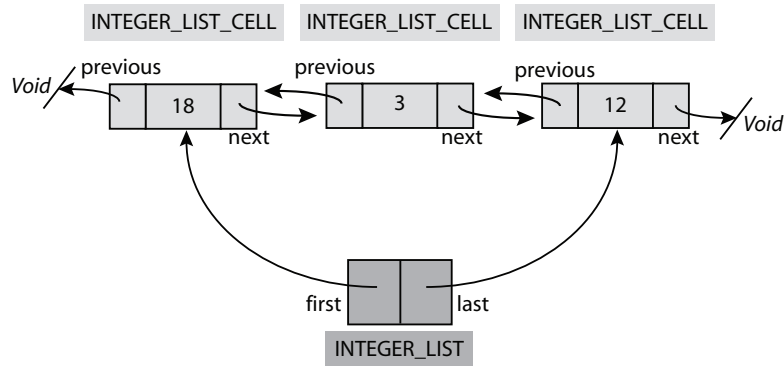


Figure 1: Doubly linked list

Listing 2: Class `INTEGER_LIST`

```

1 class INTEGER_LIST
2
3 create
4   make_empty
5
6 feature -- Initialization
7
8   make_empty is
9     -- Initialize the list to be empty.
10    do
11      first := Void
12      last := Void
13      count := 0
14    end
15
16 feature -- Access
17
18   first : INTEGER_LIST_CELL
19     -- Head element of the list, Void if the list is empty
20
21   last : INTEGER_LIST_CELL
22     -- Tail element of the list, Void if the list is empty
23
24 feature -- Measurement
25
26   count : INTEGER
27     -- Number of cells in the list
    
```

```
24 feature -- Element change
    extend (a_value: INTEGER) is
26     -- Append an integer list cell with content 'a_value' at the end of the list .
    local
28     el: INTEGER_LIST_CELL
    do
30     .....
32     .....
34     .....
36     .....
38     .....
40     .....
42     .....
44     .....
46     .....
48     .....
50     .....
52     .....
54     .....
56     .....
58     .....
60     .....
62     .....
    ensure
64     one_more: count = old count + 1
        first_set : count = 1 implies first.value = a_value
66     last_set : last.value = a_value
    end
68
feature -- Status report
70 empty: BOOLEAN is
    -- Is the list empty?
72 do
    Result := (count = 0)
74 end
```

```
76  has (a_value: INTEGER): BOOLEAN is
    -- Does the list contain a cell with value 'a_value'?
78  local
    .....
80  .....
82  .....
84  do
    .....
86  .....
88  .....
90  .....
92  .....
94  .....
96  .....
98  .....
100 .....
102 .....
104 .....
106 .....
108 .....
110 .....
112 .....
114 .....
116 .....
118 .....
120 .....
122 .....
124 .....
126  end
end
```

Listing 3: Class *INTEGER_LIST_CELL*

```
1 class INTEGER_LIST_CELL
3 create
   set_value
5
7 feature -- Access
   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
```