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`.

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

..........................................................................................

..........................................................................................

```

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

..............................................................................................
        end

feature {PERSON} -- Implementation

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

..............................................................................................

..............................................................................................

..............................................................................................

..............................................................................................
        do
            spouse := p
        ensure

..............................................................................................

..............................................................................................

..............................................................................................

..............................................................................................
        end

feature -- Basic operations

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

..............................................................................................

..............................................................................................

..............................................................................................

..............................................................................................
        do
            spouse := p
            p.accept_marriage (Current)
        ensure
..............................................................................................

..............................................................................................

```

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

.................................................................................
        end

invariant

.................................................................................

.................................................................................

.................................................................................

.................................................................................

.................................................................................

.................................................................................
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:

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

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

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

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

```
    ...........................................................................
    ...........................................................................
do
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
    ...........................................................................
ensure
    result_in_range: 0 <= Result and Result <= 9
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.

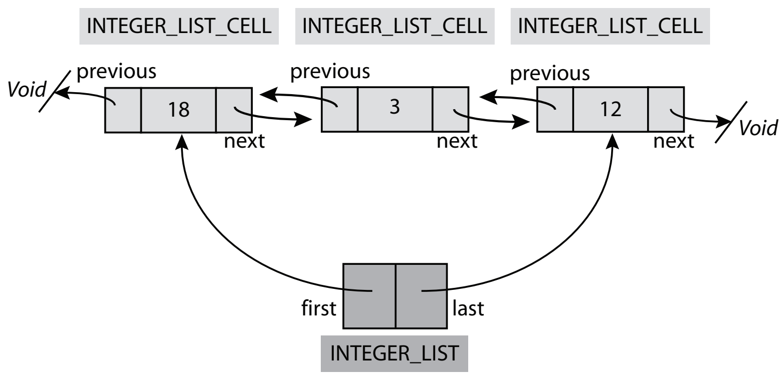

Figure 1: Doubly linked list

Read through the class *INTEGER_LIST_CELL* in Listing 2. 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 1). 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 1). 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

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: count = old count + 1
      first_set : 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
      ...............................................................................

      ...............................................................................

      ...............................................................................
    do
      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

      ...............................................................................

```

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

      ..............................................................................
    end
end
```

Listing 2: 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

feature -- Element change

  set_value (x: INTEGER) is
      -- Set 'value' to 'x'.
    do
      value := x
    ensure
      value_set: value = x
    end

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

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

end
```