Mock Exam 1

ETH Zurich

November 8,9 2010

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. A command...
   - ☐ a. call is an instruction.
   - ☐ 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. The syntax of a program...
   - ☐ a. is the set of properties of its potential executions.
   - ☐ b. can be derived from the set of its objects.
   - ☐ c. is the structure and the form of its text.
   - ☐ d. may be violated at run-time.
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. $x /= Void$ must hold.
- ☐ c. the postcondition of the creation procedure may not hold.
- ☐ d. the precondition of the creation procedure must hold.

5. Void references...

- ☐ a. cannot be the target of a successful call.
- ☐ b. are not default values for any type.
- ☐ c. indicate expanded objects.
- ☐ d. can be used to terminate linked structures (e.g. linked lists).

Solution

1. A command...

- ✓ a. call is an instruction.
- ✓ 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. The syntax of a program...

- a. is the set of properties of its potential executions.
- b. can be derived from the set of its objects.
- ✓ c. is the structure and the form of its text.
- d. may be violated at run-time.

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. $x /= Void$ must hold.
- ✓ c. the postcondition of the creation procedure may not hold.
- ✓ d. the precondition of the creation procedure must hold.

5. Void references...

   ✓ a. cannot be the target of a successful call.

   b. are not default values for any type.

   c. indicate expanded objects.

   ✓ d. can be used to terminate linked structures (e.g. linked lists).

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
```

Solution

```
class
    PERSON
```

```
create
    make

feature {NONE} -- Creation

    make (n: STRING)
            -- Create a person with a name 'n'.
        require
            n_exists: n /= Void
            n_nonempty: not n.is_empty
        do
         -- Create a copy of the argument and assign it to name
            name := n.twin
        ensure
            name_set: n.is_equal (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_exists: name /= Void
    name_nonempty: 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)

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	Example
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: $digital\_root(n) = n - n\lfloor \frac{9}{n} \rfloor$. 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
```

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 Inversion of Linked List (10 Points)

The classes *SINGLE_LINKED_LIST* [*G*] and *SINGLE_CELL* [*G*] implement a single linked list. The first cell of the list is stored in the attribute *first* of the class *SINGLE_LINKED_LIST* [*G*]. Attribute *next* of class *SINGLE_CELL* [*G*] delivers the next cell . Calling *next* on the last cell will return a *Void* reference.

Implement the feature *invert* of class *SINGLE_LINKED_LIST* [*G*], so that it inverts the order of the elements in the list. For example, inverting the list [6, 2, 8, 5] results in [5, 8, 2, 6]. **Do not** create new objects of type *SINGLE_CELL* [*G*] and also **do not** introduce any new feature in class *SINGLE_LINKED_LIST* [*G*] and *SINGLE_CELL* [*G*].

```
class
  SINGLE_LINKED_LIST [G]

feature -- Access

  first : SINGLE_CELL [G]
      -- Head element of the list, 'Void' if the list is empty

feature -- Basic operations

  invert
      -- Invert the order of the elements of the list .
      -- E.g. the list [6, 2, 8, 5] should be become [5, 8, 2, 6].
    local
      ...........................................................................

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
class
  SINGLE_CELL [G]

feature -- Access

  next: SINGLE_CELL [G]
      -- Reference to the next generic list cell of a list


feature -- Element change

  set_next (an_element: SINGLE_CELL [G])
      -- Set 'next' to 'an_element'.
    ensure
      next_set: next = an_element

end
```

Solution

```
invert
    -- Invert the order of the elements of the list .
    -- E.g. the list [6, 2, 8, 5] should be become [5, 8, 2, 6]
  local
    actual: SINGLE_CELL [G]
    next: SINGLE_CELL [G]
  do
    from
    until
      first = Void
    loop
      actual := first
      first := first.next
      actual.set_next (next)
      next := actual
    end
    first := next
end
```