Mock Exam 1

ETH Zurich

November 4, 2015

Name: 

Group: 

<table>
<thead>
<tr>
<th>Question</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question 1</td>
<td>7.5</td>
</tr>
<tr>
<td>Question 2</td>
<td>14</td>
</tr>
<tr>
<td>Question 3</td>
<td>14</td>
</tr>
<tr>
<td>Total</td>
<td>35.5</td>
</tr>
</tbody>
</table>
1 Multiple choice (7.5 points)

Put checkmarks in the checkboxes corresponding to the correct statements. There is at least one correct answer per question. A correctly checked or unchecked box is worth 0.5 points. An incorrectly checked or unchecked box is worth 0 points. Completely unanswered questions are worth 0 points.

Example:

Which of the following statements are true?

- a. The sun is a mass of incandescent gas. ☑ 0.5 points
- b. $2 \times 4 = 8$ ☐ 0 points
- c. “Rösti” is a kind of sausage. ☑ 0 points
- d. C is an object-oriented programming language. ☐ 0.5 points

1. Control structures and recursion.
   a. If we know that a loop decreases its variant and that it never goes below 5, then we know that the loop terminates. ☐
   b. The loop invariant is checked at the end of loop initialization (before entering the loop itself). ☐
   c. The loop invariant tells us how many times the loop will be executed. ☐
   d. In Eiffel a procedure can have an empty body (do end). ☐
   e. The inspect instruction can be applied to expressions of any type. ☐

2. Objects and classes
   a. All entities store references to run-time objects. ☐
   b. Different entities can reference the same object. ☐
   c. Clients of a class $X$ can see all features declared in class $X$. ☐
   d. A class needs to tell its clients whether a query is an attribute or a function. ☐
   e. Objects can be created from every class. ☐

3. Design by Contract
   a. For a feature with postcondition false, any implementation is correct. ☐
   b. Every procedure ensures that the postcondition true holds. ☐
   c. The class invariant needs to hold before every procedure call. ☐
   d. For functions, the precondition may not refer to the Result expression and the postcondition may not refer to the arguments of the function. ☐
   e. A feature with precondition false is accepted by the compiler. ☐
2 Specifying Software through Contracts (14 points)

A range of integers can be conveniently represented using the boundary values of the range, e.g., the range of integers between \( m \) and \( n \) (inclusive) can be represented using \([m, n]\). Given a range \( R \), we use \( S_R \) to denote the set of integers within \( R \), i.e.

\[
S_{[m,n]} = \{ x \mid m \leq x \leq n \}.
\]

For example, \( S_{[1,3]} = \{1, 2, 3\} \) and \( S_{[3,1]} = \emptyset \).

Listing 1 shows a class \( RANGE \), which abstracts integer ranges and provides functions that operate on them. The preconditions of the functions are already defined in the class; the function results, however, are only given in the comments in terms of the boundary values and the integer sets corresponding to the operand ranges. For example, the comment of function \( is\_equal \) stipulates that \( Result \) should be \( True \) if and only if \( Current \) and \( other \) represent the same set of integers, and the comment of function \( add \) specifies the integer set of \( Result \) should be equal to the union of the sets of \( Current \) and \( other \).

Read through the code, then complete the postconditions so that they reflect the function comments.

Please note:

- The number of dotted lines is not indicative of the number of missing contract clauses.
- You need to write \( True \) at places where you think no explicit contract is necessary: leaving a postcondition empty gives you 0 point for that section.
- The following features from class \( INTEGER \) may be useful:

```plaintext
class INTEGER

feature

max (other: INTEGER): INTEGER
    -- The greater of current integer and 'other'.

min (other: INTEGER): INTEGER
    -- The smaller of current integer and 'other'.

-- Other features omitted.

end
```

Listing 1: Class \( RANGE \)

```plaintext
note
description: "A range of integers."

class RANGE

inherit

    ANY
        redefine is_equal end

create make
```
feature \{NONE\} -- Initialization

    make (l, r : INTEGER)
    do
        left := l
        right := r
    end

feature -- Access.

    left : INTEGER
        -- Lower boundary of the range.
        -- \( S_{\text{Current}} = \{x \mid left \leq x \leq right \} \)

    right : INTEGER
        -- Upper boundary of the range.
        -- \( S_{\text{Current}} = \{x \mid left \leq x \leq right \} \)

feature -- Query

    is_equal (other : like Current): BOOLEAN
        -- Result = (\( S_{\text{Current}} = S_{\text{other}} \))
        require
            other /= Void
        ensure

    is_empty: BOOLEAN
        -- Result = (\( S_{\text{Current}} = \emptyset \))
        require
            True
        ensure

    is_sub_range_of (other : like Current): BOOLEAN
        -- Result = (\( S_{\text{Current}} \subseteq S_{\text{other}} \))
        require
            other /= Void
        ensure

    is_super_range_of (other : like Current): BOOLEAN
        -- Result = (\( S_{\text{Current}} \supseteq S_{\text{other}} \))
require
     other /= Void
ensure

left_overlaps (other: like Current): BOOLEAN
    -- Result = (left ∈ (S\text{Current} ∩ S\text{other}))
require
     other /= Void
ensure

right_overlaps (other: like Current): BOOLEAN
    -- Result = (right ∈ (S\text{Current} ∩ S\text{other}))
require
     other /= Void
ensure

overlaps (other: like Current): BOOLEAN
    -- Result = (S\text{Current} ∩ S\text{other} ≠ 0)
require
     other /= Void
ensure

feature -- Operation

add (other: like Current): RANGE
    -- S\text{Result} = (S\text{Current} ∪ S\text{other})
require
     other /= Void
     result_is_range : is_empty or other.is_empty or overlaps (other)
ensure
     Result /= Void
\begin{verbatim}
subtract (other: like Current): RANGE
   \texttt{\textbar\textbar\ S_{Result} = (S_{Current} \textbar\textbar S_{other})}
require:
   other /= Void
   result\_is\_range : not overlaps (other)
      or left\_overlaps (other) or right\_overlaps (other)
ensure
   Result /= Void
\end{verbatim}
3 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.png)

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 2: Class INTEGER_LIST

```plaintext
class INTEGER_LIST
create
make_empty
feature """"Initialization"
make_empty
  """"Initialize the list to be empty.
  first := Void
  last := Void
  count := 0
```
14  end

16 feature -- Access

18  first : INTEGER_LIST_CELL
    -- Head element of the list, Void if the list is empty

20  last : INTEGER_LIST_CELL
    -- Tail element of the list, Void if the list is empty

24 feature -- Measurement

26  count: INTEGER
    -- Number of cells in the list

28 feature -- Element change

30  extend (a_value: INTEGER)
    -- Append an integer list cell with content 'a_value' at the end of the list.

32 local
    el: INTEGER_LIST_CELL

34 do

36

38

40

42

44

46

48

50

52

54

56

58

60

62

64
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 the list empty?
do
   Result := (count = 0)
end

has (a.value: INTEGER): BOOLEAN
   -- Does the list contain a cell with value 'a.value'?
local
   .....
do
Listing 3: 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`)  

— Set ‘value’ to ‘x’.  

do  

\texttt{value} := \texttt{x}  

ensure  

\texttt{value\_set}: \texttt{value} = \texttt{x}  

end  

**set\_next** (`el`: `INTEGER_LIST_CELL`)  

— Set ‘next’ to ‘el’.  

do  

\texttt{next} := \texttt{el}  

ensure  

\texttt{next\_set}: \texttt{next} = \texttt{el}  

end  

**set\_previous** (`el`: `INTEGER_LIST_CELL`)  

— Set ‘previous’ to ‘el’.  

do  

\texttt{previous} := \texttt{el}  

ensure  

\texttt{previous\_set}: \texttt{previous} = \texttt{el}  

end  

end