Exercise 5

Hand-out: 10 May 2005
Due: 31 May 2005

Please solve this exercise within your project team.

1. Design by Contract

The principles of Design by Contract are especially crucial when reusing existing software: to be reusable, a component must be equipped with a clear specification of its working conditions through preconditions, postconditions and class invariants, which also have to be part of the software itself.

Contracts, also called assertions, are Boolean expressions stating the semantic properties of classes. They are used to express a specification. Usually, contracts include pre/postconditions, class invariants, check instructions and loop invariants/variants.

1.1 Introducing Contracts

Below, we have a class, ACCOUNT, add contracts to it and complete the comments if necessary.

```class
ACCOUNT
create
make
feature {NONE} -- Initialization
    make (an_amount: like balance) is
        -- Set ‘balance’ to ‘an_amount’.
        do
            balance := an_amount
        end
end```
feature -- Access

  balance: INTEGER
    -- Account balance

  minimum_balance: INTEGER is 1000
    -- Minimum amount of money on the account

feature -- Deposit

  deposit (an_amount: like balance) is
    -- Add 'an_amount' to 'balance'.
    do
      balance := balance + an_amount
    end

feature -- Withdrawal

  withdraw (an_amount: like balance) is
    -- Subtract 'an_amount' from current 'balance'.
    do
      balance := balance - an_amount
    end

feature -- Status report

  may_withdraw (an_amount: like balance): BOOLEAN is
    -- May 'an_amount' be withdrawn from the account?
    do
      Result := (balance - an_amount >= minimum_balance)
    end

end

1.2 Contract extraction

Here is an extract of the documentation provided with the .NET Framework for method Insert of class System.Collections.ArrayList:
**public virtual** void Insert (int index, Object value);

Inserts an element into the ArrayList at the specified index.

**Parameters**
- **index**: The zero-based index at which value should be inserted.
- **value**: The Object to insert.

**Exceptions**

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgumentOutOfRangeException</td>
<td><code>index</code> is less than zero. -or- <code>index</code> is greater than Count.</td>
</tr>
<tr>
<td>NotSupportedException</td>
<td>The ArrayList is read-only. -or- The ArrayList has a fixed size</td>
</tr>
</tbody>
</table>

**Remarks**

1. If Count already equals Capacity, the capacity of the list is doubled by automatically reallocating the internal array before the new element is inserted.
2. If Capacity is explicitly set to zero, the common language runtime sets it to the default capacity when the first element is added. The default capacity is 16.

Complete the specification with contract of this feature in Eiffel syntax below:

```eiffel
class ARRAY_LIST

feature -- Element change
insert (an_index: INTEGER; a_value: ANY) is
    -- Insert `a_value’ into the list at index `an_index’.

    require
        __?____________________________________

    do
        ......

    ensure
        __?____________________________________

end
```
2. Iterations

Assume you have an object `name_list' of type LIST [STRING]. The following code uses a loop to print all elements to the console. For simplicity let's assume also `name_list' and all of its elements are attached (are not Void).

```
local old_cursor: CURSOR
do from old_cursor := name_list.cursor
   name_list.start
   until name_list.off
loop
   print (name_list.item)
   print ("%N")
   name_list.forth
end
name_list.go_to (old_cursor)
end
```

Write down the second and more concise way to print out all elements.