Exercise 3: Objects, Design by Contract

Hand-out: 23 April 2004
Due: 30 April 2004

Please solve this exercise alone.

1. Summary: Objects, Design by Contract

1.1 Feature categories

1.2 Object cloning
Reference assignment ($a$ and $b$ of reference types): $b := a$
Object duplication (shallow): $c := \text{clone}(a)$
Object duplication (deep): $d := \text{deep}\_\text{clone}(a)$
Shallow field-by-field copy (no new object is created): $e\.\text{copy}(a)$

1.3 Design by Contract

<table>
<thead>
<tr>
<th>Routine</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>PRECONDITION</td>
<td>POSTCONDITION</td>
</tr>
<tr>
<td>Supplier</td>
<td>POSTCONDITION</td>
<td>PRECONDITION</td>
</tr>
</tbody>
</table>
Contracts are clear definitions of benefits and obligations between clients and suppliers.

They have several benefits, in particular:
- They help build correct software right from the start.
- They provide up-to-date documentation.
- They help debugging and testing.
- They provide a higher-level of discussion between programmers and managers.

The correctness of a class is defined in terms of contracts:
- For every creation procedure \( cp \): \( \{ \text{Pre}_{cp} \} \) \( \text{do} \) \( \{ \text{Post}_{cp} \text{ and } \text{INV} \} \)
- For every exported routine \( r \): \( \{ \text{INV and Pre}_r \} \) \( \text{do} \) \( \{ \text{Post}_r \text{ and } \text{INV} \} \)

2. Family tree

(This exercise in an extract of Object-Oriented Software Construction, 2\(^{nd}\) edition, by Bertrand Meyer; page 277.)

To do

Write a class PERSON covering a simple notion of person, with attributes name (of type STRING), mother, father, oldest_child, sibling (describing the next younger sibling if any), and spouse. Include routines which will find (respectively) the list of ancestors, direct cousins, cousins direct or indirect, uncles or aunts, siblings-in-laws, parents-in-laws, etc. of a given person. (Note: We only consider traditional families where partners are married; we do not consider divorce.)

Hints
- Write recursive procedures (but make sure to avoid infinite recursion where the relations, for example direct or indirect cousin, are cyclic).
- Think of adding contracts.

To hand in

Hand in the text of class PERSON.

3. Feature categories

The classes below are a simplified version of the classes appearing in the document Eiffel: The Essentials that you got during the first exercise session. The classes here do not have so-called “contracts” because you haven’t seen this notion during the lectures yet.

(WARNING: It does not mean that contracts are not important and can be omitted. On the contrary, you will see that it is an essential notion and that all your classes should express contracts.)

Let’s consider a class LIBRARY that has a list of books, each book being of type BOOK.
Here is the text of class `LIBRARY':

```plaintext
indexing

description: "Library where users can borrow books"

class

LIBRARY

create

make

feature {NONE} -- Initialization

make is

-- Create 'books'.

do

create books.make
end

feature -- Access

books: LINKED_LIST [BOOK]

-- Books available in the library

feature -- Element change

extend (a_book: BOOK) is

-- Extend 'books' with 'a_book'.

do

books.extend (a_book)
end

remove (a_book: BOOK) is

-- Remove 'a_book' from 'books'.

do

books.start

books.search (a_book)

books.remove
end

feature -- Output

display_books is

-- Display title of all 'books' in the library.

do

if books.is_empty then

io.put_string ("No book available")

else

from books.start until books.after loop

io.put_string (books.item.title)

books.forth

end

end
```

end
feature -- Basic operation

borrow_all is
  -- Borrow all 'books' available in the library.
  do
    from books.start until books.after loop
      books.item.borrow
      books.forth
    end
  end

Here is the class \textit{BOOK} of which class \textit{LIBRARY} is a client:

(Note: \textit{a\textunderscore title}: like \textit{title} means that \textit{a\textunderscore title} has the same type as \textit{title}.)

indexing

  description: "Representation of a book"

class

  BOOK

create

  make

feature {NONE} -- Initialization

  make (a\textunderscore title: like title; some\textunderscore authors: like authors) is
    -- Set `title' to `a\textunderscore title'.
    -- Set `authors' to `some\textunderscore authors'.
    do
      title := a\textunderscore title
      authors := some\textunderscore authors
    end

feature -- Access

  title: STRING
    -- Title of the book

  authors: STRING
    -- Authors of the book
    -- (if several authors, of the form: -- "first\textunderscore author, second\textunderscore author, ...")

feature -- Status report

  borrowed: BOOLEAN
    -- Is book currently borrowed (i.e. not in library)?
feature -- Basic operation

borrow is
  -- Borrow book.
  do
    borrowed := True
  end
return is
  -- Return book.
  do
    borrowed := False
  end
end

To do
Classify the features of classes LIBRARY and BOOK according to the feature classification given in the summary. Are they commands, queries, procedures, functions, attributes, routines? Explain.

<table>
<thead>
<tr>
<th>Commands</th>
<th>Queries</th>
<th>Procedures</th>
<th>Functions</th>
<th>Attributes</th>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hints
- A feature may belong to several categories.
- The number of rows in the table above is not a hint. There may be more or fewer rows in the final table.

To hand in
Hand in your classification of the features of classes LIBRARY and BOOK and corresponding explanations.

4. Design by Contract
You have seen during the last exercise session that the ADT specification of an unbounded queue (FIFO, First-In, First-Out) is the following:

TYPES
- QUEUE [G]
FUNCTIONS
• put: QUEUE \([G] \times G \rightarrow QUEUE \[G]\)
• remove: QUEUE \([G] \rightarrow QUEUE \[G]\)
• item: QUEUE \([G] \rightarrow G\)
• empty: QUEUE \([G] \rightarrow BOOLEAN\)
• new: QUEUE \([G]\)

AXIOMS
For any \(x: G, q: QUEUE \[G]\)
• item (put \((q, x)\)) = \(\begin{cases} item (q) & \text{if not empty} (q) \\ x & \text{if empty} (q) \end{cases}\)
• remove (put \((q, x)\)) = \(\begin{cases} put (remove (q), x) & \text{if not empty} (q) \\ q & \text{if empty} (q) \end{cases}\)
• empty (new)
• not empty (put \((q, x)\))

PRECONDITIONS
• remove \((q: QUEUE \[G]\)) \require not empty (q)
• item \((q: QUEUE \[G]\)) \require not empty (q)

To do
Write the interface of the Eiffel class \(QUEUE \[G]\) corresponding to the above ADT.

Hints
• \(QUEUE \[G]\) denotes a generic class. This notion will be explained in detail later in the course. Here you just need to know it represents a set of possible types (for example \(QUEUE \[[INTEGER]\], QUEUE \[[PROCESS]\], etc.) and the text of class \(QUEUE \[G]\) can use \(G\) as a common representation for concrete generic parameters such as \(INTEGER\) or \(PROCESS\). (Ask your assistant for more detail if this notion is not clear enough.)
• Do not forget to add the contracts corresponding to the ADT preconditions and axioms.

To hand in
Hand in the text of the interface of class \(QUEUE \[G]\).