Exercise 3: Objects, Design by Contract

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

Master Solution

1. Summary: Objects, Design by Contract

1.1 Feature categories

**Command**

- No result

**Procedure**

- No result

**Routine**

- Returns result

**Function**

- Returns result

**Memory**

- Memory

**Query**

- Memory

1.2 Object cloning

Reference assignment ($a$ and $b$ of reference types): $b := a$

Object duplication (shallow): $c := \text{clone}(a)$

Object duplication (deep): $d := \text{deep\_clone}(a)$

Shallow field-by-field copy (no new object is created): $e.copy(a)$

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 INV}\}
- For every exported routine \( r: \{\text{INV and Pre}_{r}\} \text{do} \{\text{Post}_{r} \text{ and INV}\}

2. Family tree

(This exercise in an extract of *Object-Oriented Software Construction, 2nd 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`.

**Solution**

```plaintext
class
  PERSON
create
  make,
  make_with_name
feature -- Initialization
```
make is
  -- Create a person with an unknown name and
  -- no relations.
  do
    name := "Unknown"
  ensure
    name_set_to_unknown: name.is_equal ("Unknown")
  end

make_with_name (a_name: STRING) is
  -- Create a person with name `a_name' and
  -- no relations.
  require
    a_name_not_void: a_name /= Void
  do
    name := a_name
  ensure
    name_set: name = a_name
  end

feature -- Access (Personal information)
  name: STRING
  -- Name of this person

feature -- Status report
  is_married: BOOLEAN is
    -- Is this person married?
    -- (i.e. Does this person have a spouse?)
    do
      Result := spouse /= Void
    ensure
      definition: Result = (spouse /= Void)
    end

feature -- Status setting (Personal information)
  set_name (a_name: STRING) is
    -- Set the name to `a_name'.
    require
      a_name_not_void: a_name /= Void
    do
      name := a_name
    ensure
      name_set: name = a_name
    end

feature -- Access (Direct relations)
  father: PERSON
  -- Father of this person

  mother: PERSON
  -- Mother of this person
sibling: PERSON  
   -- Sibling of this person

spouse: PERSON  
   -- Spouse of this person

oldest_child: PERSON  
   -- Oldest child of this person

feature -- Status setting (Direct relations)

set_father (a_father: PERSON) is
   -- Add this person as the youngest child to the
   -- family of 'a_father'.
   require
      a_father not void: a_father /= Void
      not_same: a_father /= Current
      not_descendant: not descendants.has (a_father)
      father_has_spouse: a_father.is_married
   do
      father := a_father
      mother := a_father.spouse
      father.add_child (Current)
      mother.add_child (Current)
   ensure
      father_set: father = a_father
   end

marry (a_spouse: PERSON) is
   -- Set the spouse to 'a_spouse'.
   require
      partner_not_void: a_spouse /= Void
      not_same: a_spouse /= Current
      not_ancestor_of_spouse:
         not a_spouse.ancestors.has (Current)
      spouse_not_ancestor: not ancestors.has (a_spouse)
   do
      spouse := a_spouse
      spouse.set_spouse (Current)
   ensure
      spouse_set: spouse = a_spouse
   end

feature -- Access (Groups)

ancestors: LINKED_SET [PERSON] is
   -- The ancestors of this person
   do
      create Result.make

      if father /= Void then
         Result.extend (father)
         Result.append (father.ancestors)
      end
   end
if mother /= Void then
   Result.extend (mother)
   Result.append (mother.ancestors)
end
ensure
   ancestors_not_void: Result /= Void
end

descendants: LINKED_SET [PERSON] is
   -- The descendants of this person
local
   ch: LINKED_SET [PERSON]
do
   create Result.make
   ch := children
   from
   ch.start
   until
   ch.off
   loop
      Result.extend (ch.item)
      Result.append (ch.item.descendants)
      ch.forth
   end
ensure
   descendants_not_void: Result /= Void
end

siblings: LINKED_SET [PERSON] is
   -- The siblings of this person (older and younger)
do
   if father /= Void then
      Result := father.children
      Result.prune (Current)
   else
      create Result.make
   end
ensure
   siblings_not_void: Result /= Void
   siblings_not_has_current: not Result.has (Current)
end

children: LINKED_SET [PERSON] is
   -- The children of this person
local
   cur: PERSON
do
    create Result.make
    from
    cur := oldest_child
    until
    cur = Void
    loop
    Result.extend (cur)
    cur := cur.sibling
end
ensure
  children_not_void: Result /= Void
end

direct_cousins: LINKED_SET [PERSON] is
  -- The direct cousins of this person
  local
dua: LINKED_SET [PERSON]
do
  create Result.make
dua := direct_uncles_and_aunts
  from
dua.start
  until
dua.off
  loop
    Result.append (dua.item.children)
dua.forth
  end
ensure
  direct_cousins_not_void: Result /= Void
end
cousins: LINKED_SET [PERSON] is
  -- The (direct or indirect) cousins of this person
  local
ua: LINKED_SET [PERSON]
do
  create Result.make
ua := uncles_and_aunts
  from
ua.start
  until
ua.off
  loop
    Result.append (ua.item.children)
ua.forth
  end
ensure
  cousins_not_void: Result /= Void
end
direct_uncles_and_aunts: LINKED_SET [PERSON] is
  -- The direct uncles and aunts of this person
do
  create Result.make
  if father /= Void then
    Result.append (father.siblings)
  end
  if mother /= Void then
    Result.append (mother.siblings)
  end
ensure
  direct_uncles_and_aunts_not_void: Result /= Void
end
uncles_and_aunts: LINKED_SET [PERSON] is
    -- The (direct or indirect) uncles and aunts of this
    -- person
local    dua: LINKED_SET [PERSON]
do
    create Result.make
    if father /= Void then
        dua := father.uncles_and_aunts
        from
        until    dua.off
        loop
            Result.append (dua.item.children)
            dua.forth
        end
    end
    if mother /= Void then
        dua := mother.uncles_and_aunts
        from
        until    dua.off
        loop
            Result.append (dua.item.children)
            dua.forth
        end
    end
    Result.append (direct_uncles_and_aunts)
ensure
    uncles_and_aunts_not_void: Result /= Void
end

siblings_is_law: LINKED_SET [PERSON] is
    -- The siblings-in-law of this person
local    sibs: LINKED_SET [PERSON]
do
    if is_married then
        Result := spouse.siblings
    else
        create Result.make
    end
    from
    until    sibs.off
    loop
        if sibs.item.is_married then
            Result.extend (sibs.item.spouse)
        end
        sibs.forth
    end
ensure
  siblings_in_law_not_void: Result /= Void
end

parents_in_law: LINKED_SET [PERSON] is
  -- Parents in law of this person
do
  create Result.make
  if is_married and spouse.father /= Void then
    Result.extend (spouse.father)
    Result.extend (spouse.mother)
  end
ensure
  parents_in_law_not_void: Result /= Void
end

feature {PERSON} -- Implementation

add_child (a_child: PERSON) is
  -- Add `a_child' as a child to the current list of
  -- children.
local
  cur: PERSON
do
  if oldest_child = Void then
    oldest_child := a_child
  else from
    cur := oldest_child
    until cur.sibling = Void or cur = a_child
    loop
      cur := cur.sibling
    end
    if cur /= a_child then
      cur.set_sibling (a_child)
    end
  end
end

set_sibling (a_sibling: PERSON) is
  -- Set `a_sibling' as the next younger sibling of
  -- this person.
require
  younger_siblings_not_void: a_sibling /= Void
  not_has_sibling: sibling = Void
do
  sibling := a_sibling
ensure
  sibling_set: sibling = a_sibling
end

set_spouse (a_spouse: PERSON) is
  -- Set the spouse to `a_spouse'.
do
  spouse := a_spouse
ensure
    spouse_set: spouse = a_spouse
end

invariant
    name_not_void: name /= Void
    consistent: spouse /= Void implies spouse.spouse = Current
end

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
    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

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 BOOK of which class LIBRARY is a client:
(Note: a_title: like title means that a_title has the same type as title.)

indexing

        description: "Representation of a book"

class

    BOOK

create

        make
feature {NONE} -- Initialization

make (a_title: like title; some_authors: like authors) is
    -- Set 'title' to 'a_title'.
    -- Set 'authors' to 'some_authors'.
    do
        title := a_title
        authors := some_authors
    end

feature -- Access

    title: STRING
        -- Title of the book
    authors: STRING
        -- Authors of the book
        -- (if several authors, of the form:
        -- "first_author, second_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

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.

Solution

Features of class LIBRARY:

<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>make</td>
<td>books</td>
<td>make</td>
<td></td>
<td>books</td>
<td>make</td>
</tr>
<tr>
<td>extend</td>
<td></td>
<td>extend</td>
<td></td>
<td>extend</td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td>remove</td>
<td></td>
<td>remove</td>
<td></td>
</tr>
<tr>
<td>display_books</td>
<td></td>
<td>display_books</td>
<td></td>
<td>display_books</td>
<td></td>
</tr>
<tr>
<td>borrow_all</td>
<td></td>
<td>borrow_all</td>
<td></td>
<td></td>
<td>borrow_all</td>
</tr>
</tbody>
</table>

Features of class BOOK:

<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>make</td>
<td>title</td>
<td>make</td>
<td></td>
<td>title</td>
<td>make</td>
</tr>
<tr>
<td>borrow</td>
<td>authors</td>
<td>borrow</td>
<td></td>
<td>authors</td>
<td>borrow</td>
</tr>
<tr>
<td>return</td>
<td>borrowed</td>
<td>return</td>
<td></td>
<td>borrowed</td>
<td>return</td>
</tr>
</tbody>
</table>

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
- $put: QUEUE [G] \times G \rightarrow QUEUE [G]$
- $remove: QUEUE [G] \rightarrow /\rightarrow QUEUE [G]$
- $item: QUEUE [G] \rightarrow G$
- $empty: QUEUE [G] \rightarrow BOOLEAN$
- $new: QUEUE [G]$
AXIOMS
For any $x: G, q: \text{QUEUE} [G]$
- $\text{item} \ (\text{put} \ (q, x)) = \begin{cases} \text{item} \ (q) \text{ if not empty} \ (q) \\ x \text{ if empty} \ (q) \end{cases}$
- $\text{remove} \ (\text{put} \ (q, x)) = \begin{cases} \text{put} \ (\text{remove} \ (q), x) \text{ if not empty} \ (q) \\ q \text{ if empty} \ (q) \end{cases}$
- $\text{empty} \ (\text{new})$
- $\text{not empty} \ (\text{put} \ (q, x))$

PRECONDITIONS
- $\text{remove} \ (q: \text{QUEUE} [G]) \text{ require not empty} \ (q)$
- $\text{item} \ (q: \text{QUEUE} [G]) \text{ require not empty} \ (q)$

To do
Write the interface of the Eiffel class $\text{QUEUE} [G]$ corresponding to the above ADT.

Hints
- $\text{QUEUE} [G]$ 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 $\text{QUEUE} [\text{INTEGER}], \text{QUEUE} [\text{PROCESS}]$, etc.) and the text of class $\text{QUEUE} [G]$ can use $G$ as a common representation for concrete generic parameters such as $\text{INTEGER}$ or $\text{PROCESS}$. (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.
- It may be useful to introduce queries $\text{count}$ and $\text{has}$.

To hand in
Hand in the text of the interface of class $\text{QUEUE} [G]$.

Solution

```eiffel
indexing
    description: "Representation of unbounded queues"

class interface
    \text{QUEUE} [G]

feature -- Initialization

make
    -- Create an empty queue.

ensure
    \text{is_empty}: \text{is_empty}
```
feature -- Access

    item: G
    -- Element at the front of the queue

    require
    not_empty: not is_empty

    count: INTEGER
    -- Number of elements in the queue

feature -- Status report

    is_empty: BOOLEAN
    -- Is queue empty?

    ensure
    definition: Result = (count = 0)

    has (an_element: G): BOOLEAN
    -- Does this queue contain `an_element'?

feature -- Element change

    put (an_element: G)
    -- Put `an_element' to the queue.

    ensure
    not_empty: not is_empty
    count_increased: count = old count + 1
    has_element: has (an_element)
    item_is_an_element_if_empty_before:
    old is_empty implies item = an_element

remove

    -- Remove first entered element (`item') from the queue.

    require
    not_empty: not is_empty

    ensure
    count_decreased: count = old count - 1

invariant

    count_non_negative: count >= 0

end