Programming in the large – Exercise session, 30 April 2004

Dr. Karine Arnout
Agenda for today

- A glimpse at exercise 2
  - A common mistake

- Part of exercise 3
  - Last two exercises
  - Bernd will explain 3.2 in detail next week

- Summary of the concepts learnt so far
  - Class vs. object
  - Design by Contract
  - Genericity
  - Inheritance

- Example exam questions

- Eiffel syntax and concepts
Agenda for today

- A glimpse at exercise 2
  - A common mistake

- Part of exercise 3
  - Last two exercises
  - Bernd will explain 3.2 in detail next week

- Summary of the concepts learnt so far
  - Class vs. object
  - Design by Contract
  - Genericity
  - Inheritance

- Example exam questions

- Eiffel syntax and concepts
Exercise: Write an Eiffel class corresponding to an ADT:

- **TYPES:**
  
  - `BANK_ACCOUNT`

- **FUNCTIONS:**
  
  - `new: BANK_ACCOUNT`
  - `deposit: BANK_ACCOUNT × INTEGER → BANK_ACCOUNT`
  - `withdraw: BANK_ACCOUNT × INTEGER → BANK_ACCOUNT`
  - `current_balance: BANK_ACCOUNT → INTEGER`
  - `Minimum_balance: BANK_ACCOUNT → INTEGER`
  - `holder: BANK_ACCOUNT → STRING`
  - `change_holder: BANK_ACCOUNT × STRING → BANK_ACCOUNT`

- **PRECONDITIONS:**
  
  ...
A common mistake: Write `BANK_ACCOUNT` as a generic class:

```plaintext
class BANK_ACCOUNT [G, H]
feature
deposit (a_holder: G; a_sum: H) is
  -- Add a_sum on account corresponding to a_holder.
  require
    ...
  do
    balance := balance + a_sum
  ensure
    ...
end
end
```

Would require constrained genericity...
class VECTOR [G]

feature

infix "+" (other: VECTOR [G]): VECTOR [G] is
  -- Sum of current vector and other
  require
    lower = other.lower
    upper = other.upper

local
  a, b, c: G

  do
    ... See next ...
  end

  ... Other features ...

end
The body of **infix** "+":

```plaintext
create Result.make (lower, upper)
from
  i := lower
until
  i > upper
loop
  a := item (i)
  b := other.item (i)
  c := a + b  -- Requires a "+" operation on G!
  Result.put (c, i)
  i := i + 1
end
```
Constrained genericity: The solution

- Declare class `VECTOR` as

  ```
  class VECTOR [G -> NUMERIC]
  feature
  ... The rest as before ...
  end
  ```

- Class `NUMERIC` (from the Kernel Library) provides features `infix "+"`, `infix "/"` and so on.
Agenda for today

- A glimpse at exercise 2
  - Most common mistakes

- Part of exercise 3
  - Last two exercises
  - Bernd will explain 3.2 in detail next week

- Summary of the concepts learnt so far
  - Class vs. object
  - Design by Contract
  - Genericity
  - Inheritance

- Example exam questions

- Eiffel syntax and concepts
Feature categories

Classification by role

Classification by implementation
Class \textit{LIBRARY} (1/4)

```
class LIBRARY

create make

feature \{NONE\} -- Initialization
  make is
    -- Create \textit{books}.
    do
      ...
    end

feature -- Access
  \texttt{books: LINKED\_LIST [BOOK]}
    -- Books available in the library

No result: It is a procedure/command

It's a routine

Result: It's a query

No body: It's an attribute
```
feature -- Element change

extend (a_book: BOOK) is
  -- Extend books with a_book.
  do
    ...
  end

remove (a_book: BOOK) is
  -- Remove a_book from books.
  do
    ...
  end

Two other procedures/commands
feature -- Output

display_books is
-- Display title of all books in the library.
  do
  ...
  end

feature -- Basic operation

borrow_all is
-- Borrow all books available in the library.
  do
  ...
  end

end

Two other procedures/commands
### Summary:

<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><code>make</code></td>
<td><code>books</code></td>
<td><code>make</code></td>
<td></td>
<td><code>books</code></td>
<td><code>make</code></td>
</tr>
<tr>
<td><code>extend</code></td>
<td></td>
<td><code>extend</code></td>
<td></td>
<td></td>
<td><code>extend</code></td>
</tr>
<tr>
<td><code>remove</code></td>
<td></td>
<td><code>remove</code></td>
<td></td>
<td></td>
<td><code>remove</code></td>
</tr>
<tr>
<td><code>display_books</code></td>
<td></td>
<td><code>display_books</code></td>
<td></td>
<td></td>
<td><code>display_books</code></td>
</tr>
<tr>
<td><code>borrow_all</code></td>
<td></td>
<td><code>borrow_all</code></td>
<td></td>
<td></td>
<td><code>borrow_all</code></td>
</tr>
</tbody>
</table>
class BOOK
create make
feature \{NONE\} -- Initialization
make \(a\_title: \text{like title};\ some\_authors: \text{like authors}\) is
   -- Set title to a\_title and authors to some\_authors.
   do
     ...
   end
feature -- Access
title: STRING
   -- Title of the book

Command/procedure
Attribute/query
Class BOOK (2/3)

**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*

...  

*end*

*return is*
-- Return book.

*do*

...  

*end*

Two attributes/queries

Two commands/procedures
## Summary:

<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><code>make</code></td>
<td><code>title</code></td>
<td><code>make</code></td>
<td></td>
<td><code>title</code></td>
<td><code>make</code></td>
</tr>
<tr>
<td><code>borrow</code></td>
<td><code>authors</code></td>
<td><code>borrow</code></td>
<td></td>
<td><code>authors</code></td>
<td><code>borrow</code></td>
</tr>
<tr>
<td><code>return</code></td>
<td><code>borrowed</code></td>
<td><code>return</code></td>
<td></td>
<td><code>borrowed</code></td>
<td><code>return</code></td>
</tr>
</tbody>
</table>
ADT of unbounded queues (1/2)

- **TYPES**
  - \( \text{QUEUE} [G] \)

- **FUNCTIONS**
  - \( \text{put}: \text{QUEUE} [G] \times G \rightarrow \text{QUEUE} [G] \)
  - \( \text{remove}: \text{QUEUE} [G] \rightarrow \text{QUEUE} [G] \)
  - \( \text{item}: \text{QUEUE} [G] \rightarrow G \)
  - \( \text{empty}: \text{QUEUE} [G] \rightarrow \text{BOOLEAN} \)
  - \( \text{new}: \text{QUEUE} [G] \)

- **PRECONDITIONS**
  - \( \text{remove} (q: \text{QUEUE} [G]) \text{ require not empty (q)} \)
  - \( \text{item} (q: \text{QUEUE} [G]) \text{ require not empty (q)} \)
ADT of unbounded queues (2/2)

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

QUEUE [G]

feature -- Initialization
  make
  -- Create an empty queue.
  ensure
  is_empty: 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
**Interface of class QUEUE [G] (2/3)**

**feature** -- Status report

*is_empty*: BOOLEAN

-- Is queue empty?

**ensure**

definition: \( \text{Result} = (\text{count} = 0) \)

*has* (an_element*: G): BOOLEAN

-- Does this queue contain an_element?

**feature** -- Element change

*remove*

-- Remove first entered element (*item*) from queue.

**require**

not_empty*: not *is_empty*

**ensure**

count_decreased*: \( \text{count} = \text{old count} - 1 \)
Interface of class QUEUE [G] (3/3)

\[
\text{\textit{put} (an\_element: G)}
\]

\[\text{-- Put } \text{an\_element} \text{ to the queue.}\]

\textbf{ensure}

\begin{align*}
\text{not\_empty: not } & \text{is\_empty} \\
\text{count\_increased: count} & = \text{old } \text{count} + 1 \\
\text{has\_element: has } & (\text{an\_element}) \\
\text{item\_is\_an\_element\_if\_empty\_before:} & \\
& \text{old is\_empty implies item} = \text{an\_element}
\end{align*}

\textbf{invariant}

\begin{align*}
\text{count\_non\_negative: count} & \geq 0
\end{align*}

\textbf{end}
Agenda for today

- A glimpse at exercise 2
  - Most common mistakes
- Part of exercise 3
  - Last two exercises
  - Bernd will explain 3.2 in detail next week
- Summary of the concepts learnt so far
  - Class vs. object
  - Design by Contract
  - Genericity
  - Inheritance
- Example exam questions
- Eiffel syntax and concepts
**Class vs. object**

**CLASS**

Describes the characteristics and functionalities of objects

*a_lego_block*: an element of *LEGO_BLOCK*

**LEGO_BLOCK**: a kind of object

**OBJECT**

Instance of a class
<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>
Properties of contracts

- **A contract:**
  - Binds two parties (or more): supplier, client.
  - Is explicit (written).
  - Specifies mutual obligations and benefits.
  - Usually maps obligation for one of the parties into benefit for the other, and conversely.
  - Has **no hidden clauses**: obligations are those specified.
  - Often relies, implicitly or explicitly, on general rules applicable to all contracts (laws, regulations, standard practices).
The imperative and the applicative

<table>
<thead>
<tr>
<th>do</th>
<th>ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>( balance := balance - sum )</td>
<td>( balance = \text{old balance} - sum )</td>
</tr>
<tr>
<td><strong>PREScriptive</strong></td>
<td><strong>Descriptive</strong></td>
</tr>
<tr>
<td>How?</td>
<td>What?</td>
</tr>
<tr>
<td>Operational</td>
<td>Denotational</td>
</tr>
<tr>
<td>Implementation</td>
<td>Specification</td>
</tr>
<tr>
<td>Command</td>
<td>Query</td>
</tr>
<tr>
<td>Instruction</td>
<td>Expression</td>
</tr>
<tr>
<td>Imperative</td>
<td>Applicative</td>
</tr>
</tbody>
</table>
The correctness of a class

- For every creation procedure $cp$:
  $$\{\text{Pre}_{cp}\} \text{ do}_{cp} \{\text{Post}_{cp} \text{ and } \text{INV}\}$$

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

- The worst possible erroneous run-time situation in object-oriented software development:
  - Producing an object that does not satisfy the invariant of its own class.
Genericity: Extending the notion of class

- **Abstraction**
  - SET_OF_BOOKS
- **Type parameterization**
  - LIST_OF_PEOPLE
  - LIST_OF_BOOKS
  - LINKED_LIST_OF_BOOKS
  - LIST_OF_JOURNALS
- **Specialization**
A generic class

class

STACK \[ G \] 

feature

put \( (x : G) \) is ...

item: \( G \) is ...

end

To use the class: obtain a generic derivation, e.g.

account_stack: STACK \[ ACCOUNT \]
account_stack: STACK [ACCOUNT]
point_stack: STACK [POINT]
a: ACCOUNT
p, q, r: POINT
...
point_stack.put (p)
point_stack.put (q)

r := point_stack.item
r. move (3.0, –5.0)
account_stack.put (a)
...

Chair of Software Engineering
Programming in the large – Exercise session
Inheritance

- Describe a new class as extension or specialization of an existing class. (With MULTIPLE inheritance it can be an extension of several existing classes.)

- From the module viewpoint: if B inherits from A, all the services of A are potentially available in B (possibly with a different implementation).

- From the type viewpoint: inheritance is the “is-plus-but-except” relation. If B inherits from A, whenever an instance of A is required, an instance of B will be acceptable.
What is inheritance?

class A
  feature a is
    do
      -- Some code
    end
  end
end
class B
  feature b is
    do
      -- Some code
    end
  end
end
class C
  inherit A
  feature c is
    do
      -- Some code
    end
  end
end
class D
  inherit A, B
  feature c is
do
      -- Some code
    end
  end
a is
do
    -- Some code
  end
end
b is
do
    -- Some code
  end
end
Terminology

- Parent, Child
- Ancestor, Descendant
  - The ancestors of \( B \) are \( B \) itself and the ancestors of its parents.
- Proper ancestor, Proper descendant
- Direct instance, Instance
  - The instances of \( A \) are the direct instances of its descendants.
- (Other terminology: subclass, superclass, base class)
Agenda for today

- A glimpse at exercise 2
  - Most common mistakes

- Part of exercise 3
  - Last two exercises
  - Bernd will explain 3.2 in detail next week

- Summary of the concepts learnt so far
  - Class vs. object
  - Design by Contract
  - Genericity
  - Inheritance

- Example exam questions

- Eiffel syntax and concepts
Example exam questions

- You have 10-15 minutes to read the questions carefully.

- Then, you can choose the questions you want to be discussed.
Agenda for today

- A glimpse at exercise 2
  - Most common mistakes
- Part of exercise 3
  - Last two exercises
  - Bernd will explain 3.2 in detail next week
- Summary of the concepts learnt so far
  - Class vs. object
  - Design by Contract
  - Genericity
  - Inheritance
- Example exam questions
- Eiffel syntax and concepts
Available at:
http://se.inf.ethz.ch/teaching/ss2004/0004/slides/eiffel_the_essentials.pdf

(There is a link on the course web page.)
Don’t forget the classroom exercise next week!

Have a nice weekend!