Programming in the large –
Exercise session, 11 June 2004

Dr. Karine Arnout
Agenda for today

- A glimpse at exercise 5
  - Common mistakes
- Correction of classroom exercise 2
- Project questions
Agenda for today

- A glimpse at exercise 5
  - Common mistakes
- Correction of classroom exercise 2
- Project questions
**for_all vs. do_all**

**class**

`LINEAR [G]`

**feature** -- Iteration

```
for_all (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]): BOOLEAN
-- Is `test' true for all items?
```

```
do_all (action: PROCEDURE [ANY, TUPLE [G]])
-- Apply `action' to every item.
-- Semantics not guaranteed if `action' changes the
-- structure; in such a case, apply iterator to clone of
-- structure instead.
```

```
... end

Typically used to perform the same action
on all items of a list.
```

e.g. `LINKED_LIST [G]` inherits from `LINEAR [G]`

Typically used in assertions

Chair of Software Engineering

Programming in the large – Exercise session
Agenda for today

- A glimpse at exercise 5
  - Common mistakes

- Correction of classroom exercise 2

- Project questions
Definitions

- **Polymorphism:**
  Ability for a reference to become associated at run time with instances of different classes.

- **Dynamic binding:**
  Dynamic selection of the appropriate variant of a redefined feature.
Polymorphism vs. Dynamic binding

- Polymorphism:
  
  \[ b: BOOK \quad -- \quad b \text{ declared of type } BOOK \]

  At run time, you may attach it to an object of type \( BOOK \) or to an object of any descendant type, e.g. \( TEXTBOOK \):

  \[ t: TEXTBOOK \]

  `create` \( t\).`make`

  \[ b := t \quad \text{Permitted by polymorphism} \]
Polymorphism vs. Dynamic binding

- Dynamic binding:

  \[ b : \text{BOOK} \quad \text{-- } b \text{ declared of type } \text{BOOK} \]
  \[ t : \text{TEXTBOOK} \]
  \[ \text{create } t.\text{make} \]
  \[ b := t \quad \text{-- } b \text{ is attached to an instance of} \]
  \[ \text{-- } \text{TEXTBOOK} \]

  \text{Dynamic binding ensures that the most appropriate version of }\text{print\_book}\text{ gets called, i.e. the one defined in class }\text{TEXTBOOK}, \text{called }\text{print\_textbook}.
Contracts and inheritance

- **Issues**: what happens, under inheritance, to
  - Routine preconditions and postconditions?
  - Class invariants?
Contracts and inheritance

Correct call:

```plaintext
if a1.α then
  a1.r (...)  
-- Here a1.β holds.
end
```
Assertion redeclaration rule

- When redeclaring a routine:
  - Precondition may only be kept or weakened.
  - Postcondition may only be kept or strengthened.

- Redeclaration covers both redefinition and effecting.
A simple language rule does the trick!

Redefined version may **not** have `require` or `ensure`.

May have nothing (assertions kept by default), or

```
require else new_pre
ensure then new_post
```

Resulting assertions are:

- `new_pre or else original_precondition`
- `original_postcondition and then new_post`
Invariants

- Invariant inheritance rule:
  - The invariant of a class automatically includes the invariant clauses from all its parents, “and”-ed.
  - Accumulated result visible in flat and interface forms.
Operands and options

- Two possible kinds of argument to a feature:
  - Operands: values on which feature will operate.
  - Options: modes that govern how feature will operate.

- Example: printing a real. The number is an operand; format properties (e.g. number of significant digits, width) are options.

```
print (real_value, number_of_significant_digits,
       zone_length, number_of_exponent_digits, ...)
my_window.display (x_position, y_position,
                   height, width, text, title_bar_text, color, ...)
```
Recognizing options from operands

- Two criteria to recognize an option:
  - There is a reasonable default value.
  - During the evolution of a class, operands will normally remain the same, but options may be added.
Operands and options

- The Option-Operand Separation principle:
  - The arguments of a feature should only be operands.

- Options should have default values, with procedures to set different values if requested.

- For example:

```
my_window.set_background_color("blue")
...
my_window.display
```
Advantage of options/operands separation

- It helps increase the “learnability” of a class and ease-of-use of the library:
  - If options are arguments of a routine, one needs to learn all arguments (understand what they are used for) before being able to use the routine
  - whereas it would be much easier to learn only the operands and then understand options one at a time, when one actually needs it.
Obsolete features and classes

- A central problem in the computer field: how to reconcile progress with the protection of the installed base?
- Obsolete features and classes support smooth evolution.
- In class `ARRAY`:

  ```
  enter (i: V; v: T) is
  obsolete "Use `put (value, index)`"
  do
    put (v, i)
  end
  ```
class

ARRAY_LIST [G]

obsolete

"[

Use MULTI_ARRAY_LIST instead
(same semantics, but new name
ensures more consistent terminology).
Caution: do not confuse with ARRAYED_LIST
(lists implemented by one array each).

]

inherit

MULTI_ARRAY_LIST [G]

end
The design of some software is not necessarily right the first time. The mechanism of obsolete features and classes makes it possible to tell clients that were using the obsolete version that they should change their code quickly. It makes it possible to smooth over the transition to a better design.

It gives clients a transition period to update their code and avoids breaking client code.
class BOOK
create
make
feature {NONE} -- Initialization
make is
  do
    ...
  end
feature -- Status report
  borrowed: BOOLEAN
    -- Is book currently borrowed?
feature -- Basic operation

borrow is

  -- Borrow book.
  require
  not_borrowed: not borrowed
  do
    borrowed := True
  ensure
    borrowed: borrowed
  end

return is

  -- Return book.
  require
    borrowed: borrowed
  do
    borrowed := False
  ensure
    not_borrowed: not borrowed
  end

end
class LIBRARY
inherit ANY

redefine
default_create
end

feature {NONE} -- Initialization
default_create is
do
create books.make
end

feature -- Access
books: LINKED_LIST [BOOK]
  -- All books (borrowed and not borrowed) in the library
feature -- Element change
extend (a_book: BOOK) is
  -- Extend `books' with `a_book'.
  require
    a_book_not_void: a_book /= Void
    a_book_not_in_library: not books.has (a_book)
  do
    books.extend (a_book)
  ensure
    one_more: books.count = old books.count + 1
    book_added: books.last = a_book
  end
remove (a_book: BOOK) is
  -- Remove `a_book' from `books'.
  require
    a_book_not_void: a_book /= Void
    a_book_in_library: books.has (a_book)
  do
    books.start
    books.search (a_book)
    books.remove
  ensure
    one_less: books.count = old books.count - 1
    a_book_not_in_library: not books.has (a_book)
  end
Class LIBRARY (3/3)

feature -- Output
display_books is
  -- Display title of all `books' available in the library.
do
  ...
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
ensure
  all_borrowed: books.for_all (agent {BOOK}.borrowed)...........
invariant
  books_not_void: books/= Void
  no_void_book: not books.has (Void)
Class hierarchy

```
BOOK
  make
  print_book

TEXTBOOK
  print_book
  ~ print_textbook
  print_textbook++

COMICS
  print_book
  ~ print_comics
  print_comics++
```
class BOOK
create
  make
feature -- Initialization
  make is
    -- Initialize book.
    do
      ...
    end
feature -- Output
  print_book is
    -- Print message.
    do
      io.put_string ("This is a book.%N")
    end
end
class TEXTBOOK
inherit BOOK

rename print_book as print_textbook
redefine print_textbook
end

create make

feature -- Output
  print_textbook is
    -- Print message.
    do
      io.put_string (“This is a textbook.%N”)
    end
end
class COMICS
  inherit BOOK
  rename print_book as print_comics
  redefine print_comics
end
create make
feature -- Output
  print_comics is
    -- Print message.
    do
      Precursor \{BOOK\}
      io.put_string ("This is a comics.%N")
    end
  end
end
Valid or not valid? (1/4)

Valid:

```
b: BOOK
create {TEXTBOOK} b.make
b.print_book
```

Invalid:

```
b: BOOK
create {TEXTBOOK} b.make
b.print_textbook
```

Printed message:

This is a textbook.
Valid or not valid? (2/4)

```
t: TEXTBOOK
create t.make
t.print_book
```

Invalid

```
t: TEXTBOOK
create t.make
t.print_textbook
```

Valid

Printed message:

This is a textbook.
Valid or not valid? (3/4)

```
\begin{align*}
  b &: \textit{BOOK} \\
  t &: \textit{TEXTBOOK} \\
  \text{create} \ t.\text{make} \\
  b &: := t \\
  b.\text{print}\_\text{book}
\end{align*}
```

Valid

```
\begin{align*}
  b &: \textit{BOOK} \\
  t &: \textit{TEXTBOOK} \\
  \text{create} \ t.\text{make} \\
  b &: := t \\
  b.\text{print}\_\text{textbook}
\end{align*}
```

Invalid

Printed message:

This is a textbook.
Valid or not valid? (4/4)

```
| b: BOOK        |
| c: COMICS      |
| create {COMICS} b.make |
| c /= b         |
| c.print_book   |
```

```
| b: BOOK        |
| c: COMICS      |
| create {COMICS} b.make |
| c /= b         |
| c.print_comics |
```

Printed message:
This is a book.
This is a comics.
“Way to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” [Gamma, p 315]
deferred class
  STRATEGY
feature  -- Basic operations
    do_something is
      -- Do something.
        deferred
      end
  end
end
class
  STRATEGY_A
inherit
  STRATEGY
feature -- Basic operations
  do_something is
    -- Do something.
    do
      io.put_string ("Strategy A\%N")
    end
end
class

    STRATEGY_B

inherit

    STRATEGY

feature  -- Basic operations

    do_something  is

        -- Do something.

        do

            io.put_string ("Strategy B%n")

        end

end
class CONTEXT
create make
feature {NONE} -- Initialization
make (a_strategy: like strategy) is
   -- Set strategy to a_strategy.
   require a_strategy_not_void: a_strategy /= Void
do
   strategy := a_strategy
ensure strategy_set: strategy = a_strategy
end
feature -- Basic operation
do_something is
   -- Do something. (Call algorithm corresponding to strategy.)
do
   strategy.do_something
end
Class CONTEXT (2/2)

feature -- Element change
set_strategy (a_strategy: like strategy) is
-- Set strategy to a_strategy.

require
a_strategy_not_void: a_strategy /= Void

do
strategy := a_strategy

ensure
strategy_set: strategy = a_strategy

end

feature {NONE} -- Implementation
strategy: STRATEGY
-- Strategy to be used

invariant
strategy_not_void: strategy /= Void

end
Different kinds of clients

- The **CONTEXT** has different kinds of clients:
  - **Producers (creating the context):** need to know possible concrete strategies to pass to the context (with `make` or `set_strategy`)
    ⇒ Strategy exposes implementation details to clients
  
  - **Consumers (using the context):** do not need to know about concrete strategies
    ⇒ Strategy is transparent to consumer clients
class 
  CONTEXT
create
  make
feature {NONE} -- Initialization
  make (a_procedure: like strategy_procedure) is
    -- Set strategy_procedure to a_procedure.
    require
      a_procedure_not_void: a_procedure /= Void
    do
      strategy_procedure := a_procedure
    ensure
      strategy_procedure_set:
        strategy_procedure = a_procedure
  end
feature -- Access
  strategy_procedure: PROCEDURE [ANY, TUPLE]
    -- Strategy procedure to be called
feature -- Basic operations
    do_something is
      -- Do something. (Call algorithm corresponding to strategy.)
      do
        if strategy_procedure.valid_operands ([]) then
          strategy_procedure.call ([])
        end
      end
end

feature -- Element change
    set_strategy_procedure (a_procedure: like strategy_procedure) is
      -- Set strategy_procedure to a_procedure.
      require
        a_procedure_not_void: a_procedure /= Void
      do
        strategy_procedure := a_procedure
      ensure
        strategy_procedure_set: strategy_procedure = a_procedure
      end
    invariant
      strategy_procedure_not_void: strategy_procedure /= Void
    end
Agent types: Kernel library classes

**ROUTINE**

\[\text{BASE, ARGS} \rightarrow \text{TUPLE}\]

**PROCEDURE**

\[\text{BASE, ARGS} \rightarrow \text{TUPLE}\]

**FUNCTION**

\[\text{BASE, ARGS} \rightarrow \text{TUPLE, RES}\]

*Inherits from Deferred*

*call*

*item*
deferred class
  ROUTINE [BASE_TYPE, OPEN_ARGS -> TUPLE create default_create end]
...
feature -- Basic operations
call (args: OPEN_ARGS) is
  -- Call routine with operands `args'.
  require
    valid_operands: valid_operands (args)
    callable: callable
  do
    ...
end...

feature -- Status report
valid_operands (args: OPEN_ARGS): BOOLEAN is
  -- Are `args' valid operands for this routine?
  do
    ...
end
Creating vs. calling an agent

- Writing:
  ```
  agent my_feature
  ```
  creates an agent, i.e. an object of type `ROUTINE`.

- To call an agent, one needs to execute `call` (with the proper arguments) to this `ROUTINE` object, e.g:  
  ```
  my_routine.call ([args])
  ```
class APPLICATION
create
make
feature \{NONE\} -- Initialization
make is
\hspace{1cm} -- Do something using different strategies.
  local
  a_context: CONTEXT
  do
  create a_context.make (create \{STRATEGY_A\})
a_context.do_something
create a_context.set_strategy (create \{STRATEGY_B\})
a_context.do_something
end
end
Application using the new CONTEXT

class APPLICATION
feature -- Initialization
  make is

  -- Do something using different strategies.
  local
  a_context: CONTEXT
  do
    create a_context.make (
      agent (create {STRATEGY_A}).do_something)
    a_context.do_something
    a_context.set_strategy_procedure (
      agent (create {STRATEGY_B}).do_something)
    a_context.do_something
  end

  ...
end
Naming style guidelines (1/2)

- No CamelCase, e.g. `thisIsNotProperEiffelStyle`
  - Use lower case and underscores, e.g. `this_identifier_uses_proper_naming_style`

- No upper case for feature names, e.g. `CONSTANT`
  - Use lower case, e.g. `my_constant`

- Don’t call getters `get_*`, e.g. `get_name`
  - It violates the Uniform Access principle
  - Simply use `name`

- Don’t use abbreviations
  - It makes the program unreadable (by others and by you!)
  - Use proper identifiers for feature names and arguments
Naming style guidelines (2/2)

- Use verbs for commands (no return value):
  - do_something, execute, print, search, ...

- Use nouns for queries (with return value):
  - is_borrowed, employee, student_by_name, ...

Complementary material:
OOSC2, chapter 26: “A sense of style”
Agenda for today

- A glimpse at exercise 5
  - Common mistakes
- Correction of classroom exercise 2
- Project questions
Any questions about the project so far?
Enjoy the project!

Have a nice weekend!