Classroom exercise 2

4 June 2004

Master Solution

Inheritance (4 points)

Example:

Question: Give the definition of “covariance”.

Answer: Ability to change the argument types when redefining a feature if the types conform to each other.

To do:
1. Give the definition of “polymorphism”. (2 points)

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

2. Give the definition of “dynamic binding”. (2 points)

Dynamic selection of the appropriate variant of a redefined feature.

Contracts and inheritance (4 points)

Example:

Question: Explain the rules applying to inherited preconditions.

Answer: A routine redeclaration (redefinition or effecting) may keep or weaken the preconditions. It may only use require else (for “or-ing” the preconditions); the keyword require is not valid any more.

To do:
3. Explain the rules applying to inherited postconditions. (2 points)

A routine redeclaration (redefinition or effecting) may keep or strengthen the postconditions. It may only use ensure then (for “and-ing” the postconditions); the keyword ensure is not valid any more.
4. Explain the rules applying to inherited class invariants. (2 points)

Invariants of all parents of a class apply to the class itself.

**Design principles (8 points)**

**Example:**

**Question:** What is the Command-Query separation principle?

**Answer:** A command (procedure) does something but does not return a result. A query (function or attribute) returns a result but does not change the state; they should not produce abstract side effects.

**To do:**

5. What is the Option-Operand separation principle? (2 points)

The arguments of a routine should only include operands (no option). An argument is an option if, assuming the client had not supplied its value it would not have been possible to find a reasonable default. In the evolution of a class, arguments tend to remain the same, but options may be added and removed. In other words, operands are values on which a feature will operate; options are modes that govern how a feature will operate.

**Example:**

**Question:** Give a typical example of the Command-Query separation principle.

**Answer:**

```pascal
class LIST [G]

…

feature -- Basic operation
    search (an_item:G): INTEGER is
    -- Search for `an_item` and return the index in the list
    -- if found; zero otherwise.
        do
        …
    end

end
```

The feature `search` is a command; it should not return a result. A proper implementation of the class `LIST [G]` would be to make the index available as an independent query `found_index`

**To do:**

6. Give a typical example of the Option-Operand separation principle. (4 points)
Typical examples:
- Displaying a window:
  
  \[ \text{my\_window}\text{.display (x\_position, y\_position, height, width, color, \ldots)} \]

  All arguments are options. It should be simply:

  \[ \text{my\_window}\text{.display} \]

- Printing a real:

  \[ \text{print (real\_value, number\_of\_significant\_digits, zone\_length, \ldots)} \]

  The number is an operand; the other arguments are options. It should be:

  \[ \text{print (real\_value)} \]

Example:
Question: What is the advantage of a design applying the Command-Query separation principle?

Answer: Forbidding side-effects in functions (queries) allow keeping the software in line with its mathematical foundation, an ADT. Maintaining a clear distinction between commands and queries ensures that talking about “functions” in software does not betray the meaning of this term in ordinary mathematics. This principle may be expressed informally as “asking a question should not change the answer”.

To do:
7. What is the advantage of a design applying the Option-Operand separation principle? (2 points)

It helps increase the “learnability” of a class and ease-of-use of the library. Indeed, 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 classes and features (6 points)

Example:
Question: What is an obsolete class?

Answer: An obsolete class is a class with an \texttt{obsolete} clause.

To do:
8. What is an obsolete feature? (2 points)
An obsolete feature is a feature with an **obsolete** clause.

### Example:

**Question:** Give an example of an obsolete class.

**Answer:**
```plaintext
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
```

### To do:

9. Give an example of an obsolete feature. (2 points)

```plaintext
enter (i: INTEGER; v: G) is
    -- Add element `v' a position `i'.
    obsolete "Use put (v, i) instead"
    require correct_index (i)
    do
        put (v, i)
    ensure added: entry (i) = v
end
```

10. What is an obsolete feature useful for? (2 points)

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.

### Design by Contract (24 points)

Consider the following class `BOOK':
To do:
11. Add contracts (preconditions, postconditions, class invariants) to the following class `LIBRARY`. (Note: There is exactly one assertion missing per dotted line; you may write directly on the exercise sheet.) (24 points)
feature -- Element change
extend (a_book: BOOK) is
  -- Extend 'books' with 'a_book'.
  require
  .................................................................
  .................................................................
  do
  books.extend (a_book)
  ensure
  .................................................................
  .................................................................
end

remove (a_book: BOOK) is
  -- Remove 'a_book' from 'books'.
  require
  .................................................................
  .................................................................
  do
  books.start
  books.search (a_book)
  books.remove
  ensure
  .................................................................
  .................................................................
end

feature -- Output
display_books is
  -- Display title of all 'books' available in the library.
  do
  if books.is_empty then
    io.put_string ("No book available at the moment")
  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
  ensure
  .................................................................
end

invariant
  .................................................................
  .................................................................
end
class LIBRARY
inherit ANY

redefine
  default_create
end

feature {NONE} -- Initialization
default_create 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'.
    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
    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
    book_not_in_library: not books.has (a_book)
  end

feature -- Output
  display_books is
    -- Display title of all `books' available in the library.
    do
      if books.is_empty then
        io.put_string ("No book available at the moment")
      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

ensure
    all_borrowed: books.for_all (agent {BOOK}.is_borrowed)
end

invariant
    books_not_void: books /= Void
    no_void_book: not books.has (Void)
end

Inheritance (24 points)

Consider the following inheritance hierarchy:

and the corresponding class texts:

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

TEXTBOOK
print_book

COMICS
print_book

print_textbook
print_comics
print_textbook++

print_comics++
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

Example:

Question1: Is the following code valid? Explain why or why not.

```
  b: BOOK
  create b.make
  b.print_book
```

Answer1: Yes, because `b` is of type `BOOK` and class `BOOK` has a feature `print_book`.

Question2: The code presented in question 1 is valid. What message is printed when executing this code?

Answer2: This is a book.

To do:

12. Is the following code valid? Explain why or why not. (2 points)
Yes because \( b \) is declared of type \( BOOK \) and class \( BOOK \) has a feature \( print\_book \).
For the compiler, whether \( b \) is created as a direct instance of \( BOOK \) or of \( TEXBOOK \) does not matter; it is the declared type that matters.

13. Is the following code valid? Explain why or why not. (2 points)
\[
b: \text{BOOK} \\
\text{create} \ \{\text{TEXTBOOK}\} \ b.\text{make} \\
b.\text{print\_book}
\]

No, because \( b \) is declared of type \( BOOK \) and class \( BOOK \) does not have a feature \( print\_textbook \). The fact that \( b \) is attached at run time to a direct instance of \( TEXBOOK \) and class \( TEXTBOOK \) has a feature \( print\_textbook \) does not matter; the compiler only looks at the declared type.

14. One of the code samples presented in question 12 or 13 is valid. What message is printed when executing this code? (2 points)

The code presented in question 12 is valid:
\[
b: \text{BOOK} \\
\text{create} \ \{\text{TEXTBOOK}\} \ b.\text{make} \\
b.\text{print\_book}
\]

When executing this code, the following message appears: This is a textbook.
Indeed, at run time, \( b \) is attached to a direct instance of \( TEXTBOOK \). Thus, the version of \( print\_book \) that gets executed is the version from class \( TEXTBOOK \), meaning feature \( print\_textbook \) of class \( TEXTBOOK \) (because of the \textbf{rename} clause).

15. Is the following code valid? Explain why or why not. (2 points)
\[
t: \text{TEXTBOOK} \\
\text{create} \ t.\text{make} \\
t.\text{print\_book}
\]

No it is not valid because \( t \) is declared of type \( TEXTBOOK \) and class \( TEXTBOOK \) renames the feature \( print\_book \) inherited from \( BOOK \) as \( print\_textbook \); thus there is no feature \( print\_book \) in class \( TEXTBOOK \) and \( t.\text{print\_book} \) is invalid.

16. Is the following code valid? Explain why or why not. (2 points)
\[
t: \text{TEXTBOOK} \\
\text{create} \ t.\text{make} \\
t.\text{print\_textbook}
\]

Yes because \( t \) is declared of type \( TEXTBOOK \) and class \( TEXTBOOK \) renames the feature \( print\_book \) inherited from \( BOOK \) as \( print\_textbook \); thus there is a feature \( print\_textbook \) applicable to \( t \) and the above code is valid.
17. One of the code samples presented in question 15 or 16 is valid. What message is printed when executing this code? (2 points)

The code presented in question 16 is valid:
```plaintext
t: TEXTBOOK
create t.make
t.print_textbook
```
When executing this code, the following message appears: This is a textbook.

18. Is the following code valid? Explain why or why not. (2 points)
```plaintext
b: BOOK
t: TEXTBOOK
create t.make
b := t
b.print_book
```
Yes because `t` is a direct instance of TEXTBOOK and type TEXTBOOK conforms to BOOK (because TEXTBOOK inherits from BOOK); thus the assignment `b := t` is valid; then, `b` is of type BOOK and class BOOK has a feature `print_book`; thus `print_book` can be applied to `b`, and the above code is valid.

19. Is the following code valid? Explain why or why not. (2 points)
```plaintext
b: BOOK
t: TEXTBOOK
create t.make
b := t
b.print_textbook
```
No it is not valid because `t`, even if it is a direct instance of TEXTBOOK, is assigned to `b`, which is of type BOOK, and class BOOK does not have any feature `print_textbook` (it only has `print_book`); thus `b.print_textbook` is invalid.

20 One of the code samples presented in question 18 or 19 is valid. What message is printed when executing this code? (2 points)

The code presented in question 18 is valid:
```plaintext
b: BOOK
t: TEXTBOOK
create t.make
b := t
b.print_book
```
When executing this code, the following message appears: This is a textbook. (Indeed, `b` is of dynamic type TEXTBOOK – because it results from the assignment of `t` to `b` – and thanks to dynamic binding, it is the version of `print_book` of class TEXTBOOK – meaning `print_textbook` – that will be called; hence the message.)
21. Is the following code valid? Explain why or why not. (2 points)
   
   ```
   b: BOOK
   c: COMICS
   create {COMICS} b.make
   c ?= b
   c.print_book
   ```
   
   No it is not valid. Indeed, \( b \) is a direct instance of \( COMICS \); hence the assignment attempt \( c \ ?= b \) will work and \( c \) will get attached to \( b \), meaning that \( c \) is a (non-void) direct instance of \( COMICS \) and class \( COMICS \) does not have a feature \( \text{print\_book} \) (it is renamed as \( \text{print\_comics} \)); thus the code \( c.\text{print\_book} \) is invalid.

22. Is the following code valid? Explain why or why not. (2 points)
   
   ```
   b: BOOK
   c: COMICS
   create {COMICS} b.make
   c ?= b
   c.print_comics
   ```
   
   Yes because \( b \) is a direct instance of \( COMICS \) and the assignment attempt \( c \ ?= b \) will work; hence \( c \) will become a (non-void) direct instance of \( COMICS \) and class \( COMICS \) has a feature \( \text{print\_comics} \); thus the above code is valid.

23. One of the code samples presented in question 21 or 22 is valid. What message is printed when executing this code? (2 points)

   The code presented in question 22 is valid:
   ```
   b: BOOK
   c: COMICS
   create {COMICS} b.make
   c ?= b
   c.print_comics
   ```

   When executing this code, the following message appears:
   
   This is a book.
   This is a comics.

   (Because \( \text{print\_comics} \) first calls its Precursor feature, meaning \( \text{print\_book} \) from class \( BOOK \), which displays “This is a book.%N”, and then prints “This is a comics.”.)

Using agents (30 points)

Consider the following class diagram:
and the corresponding class texts:

```plaintext
defered class STRATEGY
  feature -- Basic operations
    do_something is
      -- Do something.
      deferred
    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 operations
do_something is
  -- Do something. (Call algorithm corresponding to `strategy'.)
do
    strategy.do_something
end

feature -- Access
strategy: STRATEGY
  -- Strategy to be applied

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

invariant
  strategy_not_void: strategy /= Void
end

class APPLICATION
create
  make
feature {NONE} -- Initialization
make is
  -- Do something using different strategies.
local
  a_context: CONTEXT
do
    create a_context.make (create {STRATEGY_A})
    a_context.do_something
    a_context.set_strategy (create {STRATEGY_B})
    a_context.do_something
end

end

It corresponds to a typical Eiffel implementation of the “Strategy” pattern. The application (corresponding to class APPLICATION) creates a context (corresponding to class CONTEXT) with a certain strategy (corresponding to the class STRATEGY and its descendants STRATEGY_A and STRATEGY_B).

The goal of this exercise is to use agents instead of inheritance to implement the Strategy pattern.

To do:
24. Rewrite the class CONTEXT to have an attribute strategy_procedure of type PROCEDURE [ANY, TUPLE] (instead of an attribute strategy of type STRATEGY). (Don’t forget to update the implementation of feature do_something.) (10 points)
class CONTEXT
create
make
feature {NONE} -- Initialization
make (a_procedure: like strategy_procedure) is
  require
    a_procedure_not_void: a_procedure /= Void
  do
    strategy_procedure := a_procedure
  ensure
    strategy_procedure_set: strategy_procedure = a_procedure
end
feature -- Basic operations
do_something is
  require
    a_procedure_not_void: a_procedure /= Void
  do
    if strategy_procedure.valid_operands ([]) then
      strategy_procedure.call ([])
    end
end
feature -- Access
strategy_procedure: PROCEDURE [ANY, TUPLE]
  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

25. Rewrite the class APPLICATION and possibly the other classes in order to use this new implementation of class CONTEXT (i.e. using agents instead of a STRATEGY object). (20 points)

One possibility is to only change the class APPLICATION as follows:

class APPLICATION
create
make
feature {NONE} -- Initialization
make is
  require
    a_procedure_not_void: a_procedure /= Void
  do
    strategy_procedure := a_procedure
  ensure
    strategy_procedure_set: strategy_procedure = a_procedure
end

local
  a_context: CONTEXT
do
  create a_context.make (agent (create {STRATEGY_A}).do_something)
  a_context.do_something
  a_context.set_strategy (agent (create {STRATEGY_B}).do_something)
  a_context.do_something
end