Object-Oriented Software Construction

Bertrand Meyer
Lecture 14:
Some design principles
Simplicity

- Command-query separation principle:
  - Clear, understandable interfaces

- Systematic naming conventions

- Operand-option separation:
  - Dramatically simplified feature interfaces
nonlinear_ode

(equation_count: in INTEGER;
epsilon: in out DOUBLE;
func: procedure
  (eq_count: INTEGER; a: DOUBLE;
eps: DOUBLE; b: ARRAY [DOUBLE];
cm: pointer Libtype);
left_count, coupled_count: INTEGER ...)

[And so on. Altogether 19 arguments, including:

- 4 in out values;
- 3 arrays, used both as input and output;
- 6 functions, each with 6 or 7 arguments, of which 2 or 3 arrays!]
The EiffelMath routine

... Set up the non-default values ...

e.solve

... Solve the problem, recording the answer in \( x \) and \( y \) ...
The Consistency Principle

- All the components of a library should proceed from an overall coherent design, and follow a set of systematic, explicit and uniform conventions.

- Two components:
  - Top-down and deductive (the overall design).
  - Bottom-up and inductive (the conventions).
Abstraction and objects

- Not all classes describe “objects” in the sense of real-world things.

- Types of classes:
  - Analysis classes – examples: AIRPLANE, CUSTOMER, PARTICLE.
  - Design classes – examples: STATE, COMMAND, HANDLE.
  - Implementation classes – examples: ARRAY, LINKED_LIST.

- More important than the notion of object is the concept of abstract data type (or “data abstraction”).

- Key to the construction of a good library is the search for the best abstractions.
Avoiding improper classes

- A few danger signals:
  - A class whose name is a verb in the imperative form, e.g. \textit{ANALYZE}. (Exception: command classes.)
  - A class with no parent and just one exported routine.
  - A class that introduces or redeclares no feature. (Purely taxonomical role only.) \textit{TAXOMANIA}

- Names that warrant some attention: “er” names, e.g. \textit{ANALYZER}. 
Active data structures

- Old interface for lists:
  \[ l.\text{insert} \ (i, \ x) \]
  \[ l.\text{remove} \ (i) \]
  \[ \text{pos} := l.\text{search} \ (x) \]
  \[ l.\text{insert\_by\_value} \ (...) \]
  \[ l.\text{insert\_by\_position} \ (...) \]
  \[ l.\text{search\_by\_position} \ (...) \]

- New interface:
  Queries:
  \[ l.\text{index} \]
  \[ l.\text{item} \]
  \[ l.\text{before} \]
  \[ l.\text{after} \]

  Commands:
  \[ l.\text{start} \]
  \[ l.\text{forth} \]
  \[ l.\text{finish} \]
  \[ l.\text{back} \]
  \[ l.\text{go} \ (i) \]

\[ j := l.\text{search} \ (x); \]
\[ l.\text{insert} \ (j + 1, \ y) \]

--- Typical sequence:
A list seen as an active data structure

- before
- item
- after
- back
- forth
- index
- count
An object as machine: “iterators”
Command-Query separation principle

- 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.

- This principle excludes many common schemes, such as using functions for input (e.g. C’s `getint` or equivalent).
Referential transparency

- If two expressions have equal value, one may be substituted for the other in any context where that other is valid.
  - If $a = b$, then $f(a) = f(b)$ for any $f$.
  - Prohibits functions with side effects.
  - Also:
    - For any integer $i$, normally $i + i = 2 \times i$;
    - But even if $\text{getint}() = 2$, $\text{getint}() + \text{getint}()$ is usually not equal to 4.
Command-query separation

- Input mechanism (instead of \( n := getint() \)):

\[
\begin{align*}
io\text{-}read\_integer \\
n := io\text{-}last\_integer
\end{align*}
\]
Include as many visible assertions as possible:

- Assertions help design the libraries right.
- Preconditions help find errors in client software.
- Library documentation fundamentally relies on assertions (interface forms).

\[
\begin{align*}
l.i & .insert \ (x, \ j + \ k + 1) \\
insert \ (x: \ G; \ i: \ INTEGER) \\
require \ \\
i & \geq 0 \\
i & \leq count + 1
\end{align*}
\]
Designing for consistency: An example

- Describing active structures properly: can after also be before?

- Symmetry:

<table>
<thead>
<tr>
<th>start</th>
<th>finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>forth</td>
<td>back</td>
</tr>
<tr>
<td>after</td>
<td>before</td>
</tr>
</tbody>
</table>

- For symmetry and consistency, it is desirable to have the invariant properties.

\[
\begin{align*}
  \text{after} & = (index = count + 1) \\
  \text{before} & = (index = 0)
\end{align*}
\]
Designing for consistency (cont’d)

- Typical iteration:

  ```
  from
  start
  until
  after
  loop
  some_action (item)
  forth
  end
  ```

- Conventions for an empty structure?
  - `after` must be true for the iteration.
  - For symmetry: `before` should be true too.

- But this does not work for an empty structure (`count = 0`, see invariant A): should `index` be 0 or 1?
To obtain a consistent convention we may transform the invariant into:

\[
\begin{align*}
\text{after} & = (\text{is_empty} \text{ or } (\text{index} = \text{count} + 1)) \\
\text{before} & = (\text{is_empty} \text{ or } (\text{index} = 0))
\end{align*}
\]

-- Hence: \text{is_empty} = (\text{before} \text{ and } \text{after})

Symmetric but unpleasant. Leads to frequent tests of the form

\[
\text{if after and not is_empty then ...}
\]

instead of just

\[
\text{if after then ...}
\]
Introducing sentinel items

- Invariant (partial):
  
  \[ 0 \leq index \]
  
  \[ index \leq count + 1 \]
  
  \[ \text{before} = (index = 0) \]
  
  \[ \text{after} = (index = count + 1) \]
  
  not (after and before)

```
before  not after
not after  not before
1 \leq index; index \leq count
```

Valid cursor positions
The case of an empty structure

Valid cursor positions

<table>
<thead>
<tr>
<th>0</th>
<th>1 (i.e. ( \text{count} + 1 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>after</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
</tr>
</tbody>
</table>

(i.e. count + 1)
Can after also be before?

- Lessons from an example; General principles:
  - Consistency
    - A posteriori: “How do I make this design decision compatible with the previous ones?”.  
    - A priori: “How do I take this design decision so that it will be easy – or at least possible – to make future ones compatible with it?”.  
  - Use assertions, especially invariants, to clarify the issues.
  - Importance of symmetry concerns (cf. physics and mathematics).
  - Importance of limit cases (empty or full structures).  

Chair of Software Engineering

OOSC - Summer Semester 2004
Abstract preconditions

- Example (stacks):

```
put is
  require not full
do ...
ensure ...
end
```
Would you rather buy or inherit?

- Inheritance is the “is-a” relation.
- In some cases “is-a” is clearly not applicable.
- Implementation can be a form of “is-a”
  - Example: the marriage of convenience.
When inheritance won’t do

- From: Ian Sommerville: *Software Engineering*, 4th edition, Addison-Wesley:
  - Multiple inheritance allows several objects to act as base objects and is supported in object-oriented languages such as Eiffel (Meyer, 1988). The characteristics of several different object classes can be combined to make up a new object.
  - For example, say we have an object class **CAR** which encapsulates information about cars and an object class **PERSON** which encapsulates information about people. We could use both of these to define a new object class **CAR-OWNER** which combines the attributes of **CAR** and **PERSON**.
  - Adaptation through inheritance tends to lead to extra functionality being inherited, which can make components inefficient and bulky.
  - Where is the “is-a”?
When inheritance won’t do (cont’d)

- **PERSON**
- **CAR**
- **CAR_OWNER**
The proper structure

PERSON

CAR

CAR_OWNER
"He has a head like an Austin Mini with the doors open."
(From: *The Dictionary of Aussie Slang*, Five-Mile Press, Melbourne, Australia.)
Would you rather buy or inherit?

- Except for polymorphic uses, inheritance is never required:
  - Rather than having $B$ inherit from $A$ you can always have $B$ include an attribute of type $A$ (or expanded $A$) – except if an entity of type $A$ may have to represent values of type $B$. 

(B)  

(A)
(1) Every software engineer is an engineer.
(2) Every software engineer has a part of himself which is an engineer.

But:

TO HAVE IS NOT ALWAYS TO BE!
Would you rather buy or inherit?

- A case in which having is not being (i.e. “client” is OK but not inheritance):
  - Every object of type $B$ has a component of type $A$, BUT that component may need to be replaced during the object’s lifetime.

- Use the client relation instead:

  ```
  class WINDOW inherit
    GENERAL_WINDOW
    WINDOW_IMPLEMENTATION
  feature
    ...
  end
  ```
class WINDOW inherit GENERAL_WINDOW

feature

handle: TOOLKIT

... set_handle (t: TOOLKIT) is do handle := t end ...

end
Handles (cont’d)

class TOOLKIT_FACILITIES feature
  impl: IMPLEMENTATION is
  once
       create Result
  end

set_handle (t: TOOLKIT) is
  do
       impl.set_handle (t)
  end
end

This is a class meant to be inherited by classes needing its facilities.
End of lecture 14