Object-Oriented Design principles
Designing for reuse

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 number
The number is an operand; format properties (e.g. number of significant digits, width) are options.

Examples:
(Non-O-O) print (real_value, number_of_significant_digits, zone_length, number_of_exponent_digits, ...)
(O-O) 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.

The Option-Operand Principle

Only operands should appear as arguments of a feature

Option values:
• Defaults (specified universally, per type, per object)
• To set specific values, use appropriate "setter" procedures

Example:
my_window.set_background_color ("blue")
...
my_window.display

Useful checklist for options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Set</th>
<th>Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window color</td>
<td>White</td>
<td>set_background_color</td>
<td>background_color</td>
</tr>
<tr>
<td>Hidden?</td>
<td>No</td>
<td>set_visible, set_hidden</td>
<td>hidden</td>
</tr>
</tbody>
</table>

Typical API in a traditional library (NAG)

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 array!]
The EiffelMath routine

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

\texttt{e.solve}

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

Naming (classes, features, variables...)

Traditional advice (for ordinary application programming):

- Choose meaningful variable names!

Naming example, original choices

<table>
<thead>
<tr>
<th>Class</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>enter entry</td>
</tr>
<tr>
<td>STACK</td>
<td>push top pop</td>
</tr>
<tr>
<td>QUEUE</td>
<td>add oldest remove_oldest</td>
</tr>
<tr>
<td>HASH_TABLE</td>
<td>insert value delete</td>
</tr>
</tbody>
</table>

Naming example, revised choices

<table>
<thead>
<tr>
<th>Class</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>put item</td>
</tr>
<tr>
<td>STACK</td>
<td>put item remove</td>
</tr>
<tr>
<td>QUEUE</td>
<td>put item remove</td>
</tr>
<tr>
<td>HASH_TABLE</td>
<td>put item remove</td>
</tr>
</tbody>
</table>

Naming rules

Achieve consistency by systematically using a set of standardized names.

Emphasize commonality over differences.

Differences will be captured by:

- Signatures (number and types of arguments and result).
- Assertions.
- Comments.

Some standard names

Queries (non-boolean):
- count, capacity
- item
- to_external, from_external

Commands:
- put, extend, replace, force
- wipe_out, remove, prune
- make -- For creation

Boolean queries:
- writable, readable
- is_empty, is_full

-- Usual invariants:
- \( 0 \leq \text{count} \leq \text{capacity} \)
- \( \text{is_empty} = (\text{count} = 0) \)
- \( \text{is_full} = (\text{count} = \text{capacity}) \)

-- Some rejected names:
- if \( s \).addable then
- \( s \).add(v) end
- if \( s \).deletable then
- \( s \).delete(v) end
Grammatical rules

Procedures (commands): verbs in infinitive form.
Examples: make, put, display.

Boolean queries: adjectives
Example: full (older convention)
Now recommended: is_full, is_first.
Convention: Choose form that should be false by default
Example: is_errorous.
This means that making it true is an event worth talking about.

Other queries: nouns or adjectives.
Examples: count, error_window.
Do not use verbs for queries, in particular functions: this goes with
Command-Query Separation Principle
Examples: next_item, not get_next_item

Feature categories

Standard categories (the only ones in EiffelBase):

- Access
- Measurement
- Comparison
- Status report
- Transformation

Basic commands

- Status setting
- Cursor movement
- Element change
- Removal
- Resizing
- Transformation

- Inapplicable
- Implementation
- Miscellaneous

Obsolete features and classes

A constant problem in information technology:
How do we reconcile progress with the need to protect
the installed base?

Obsolete features and classes support smooth evolution.

In class ARRAY:

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

Obsolete classes

```eiffel
class ARRAY_LIST [G]
obsolete
  Use MULTI_ARRAY_LIST instead
  (same semantics, but new name
  ensures more consistent terminology).
  Caution: do not confuse with ARRANGED_LIST
  (lists implemented by one array each).
"
inherit
  MULTI_ARRAY_LIST [G]
end
```

Complementary material

OOSC2:
- Chapter 22: How to find the classes
- Chapter 23: Principles of class design
End of lecture 18