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) \texttt{print (real_value, number\_of\_significant\_digits, zone\_length, number\_of\_exponent\_digits, ...)}
(O-O) \texttt{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
```

Operands and options

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</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 arrays!]
The EiffelMath routine

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

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</td>
</tr>
<tr>
<td>STACK</td>
<td>put</td>
</tr>
<tr>
<td>QUEUE</td>
<td>put</td>
</tr>
<tr>
<td>HASH_TABLE</td>
<td>put</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, extendable, prunable
- is_empty, is_full

-- Usual invariants:
- 0 <= count; count <= capacity
- is_empty = (count = 0)
- is_full = (count = capacity)

Some rejected names:
- if s.addable then add (v)
- if s.deletable then delete (v)
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_an_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
Example: next_item, not get_next_item

Feature categories

class C
inherit ...
feature -- Category 1 ...
  ... Feature declarations
feature (A, B) -- Category 2 ...
  ... Feature declarations
feature (NONE) -- Category n ...
  ... Feature declarations
invariant ...
end

Feature categories

Standard categories (the only ones in EiffelBase):

- Creation
  - Access
  - Measurement
  - Comparison
  - Status report

- Basic queries

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

- Inapplicable
- Implementation
- Miscellaneous

- Transformation
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:

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enter (i: V; v: T) is
  obsolete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

Obsolete classes

```
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
```

Complementary material

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