Designing for reuse

“Formula-1 programming”

The opportunity to get things right
Typical API in a traditional library (NAG)

```plaintext
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

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

e.solve
  ... Answer available in e.x and e.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).
The key to building a library

Devising a theory of the underlying domain

Some of the theory behind EiffelBase

Active data structures

Old interface for lists:
- l.insert (i, x)
- l.remove (i)
- pos := l.search (x)
- l.insert_by_value (x)
- l.insert_by_position (x)
- l.search_by_position (x)

New interface:
- Queries:
  - l.index
  - l.item
  - l.before
  - l.after
- Commands:
  - l.start
  - l.forth
  - l.finish
  - l.back
  - l.go (i)
  - l.search (x)
  - l.put (x)
  - l.remove

Typical sequence:
- j := l.search (x);
- l.insert (j + 1, y)
A list seen as an active data structure

before               item               after
back                count
forth
index

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 \( n := \text{getint}() \)):

```plaintext
io.read_integer
n := io.last_integer
```

---

**Libraries and assertions**

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).

**APPLICATION**

```
insert(x, j + k + 1)
```

**LIBRARY**

```
insert(x: G; i: INTEGER)
require
i >= 0
i <= count + 1
```

---

**Designing for consistency: An example**

Describing active structures properly: can \( \text{after} \) also be \( \text{before} \)?

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>
```

Valid cursor positions:

```
valid cursor positions
```

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

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

Typical iteration:

```plaintext
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:

```plaintext
after = (is_empty or (index = count + 1))
before = (is_empty or (index = 0))
```

-- Hence: `is_empty = (before and after)`

Symmetric but unpleasant, leads to frequent tests

```plaintext
if after and not is_empty then ...  
instead of just
if after then ...
```

Introducing sentinel items

Invariant (partial):

```plaintext
0 <= index
index <= count + 1
```

```plaintext
A
before = (index = 0)
```

```plaintext
after = (index = count + 1)
```

```plaintext
not (after and before)
```

---

Valid cursor positions
The case of an empty structure

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).

Abstract preconditions

Example (stacks):

```plaintext
put is require not full
do...
ensure...
end...
```
How big should a class be?

The first question is how to measure class size. Candidate metrics:

- Source lines
- Number of features.

For the number of features the choices are:

- With respect to information hiding:
  - Internal size: includes non-exported features.
  - External size: includes exported features only.

- With respect to inheritance:
  - Immediate size: includes new (immediate) features only.
  - Flat size: includes immediate and inherited features.
  - Incremental size: includes immediate and redeclared features.

The features of a class

- Immediate
- Redefined
- Redeclared
- Changed
- Inherited
- Effected
- Kpt
- Unchanged
- New in class
- From parent
- Was deferred
- Had an implementation

Most useful measure is incremental size. Easy to measure.

Incremental size

- Immediate
- Redefined
- Redeclared
- Changed
- Inherited
- Effected
- Kpt
- Unchanged
- New in class
- From parent
- Was deferred
- Had an implementation

Most useful measure is incremental size. Easy to measure.
The shopping list approach

If a feature may be useful, it probably is.
An extra feature cannot hurt if it is designed according to the spirit of the class (i.e. properly belongs in the underlying abstract data type), is consistent with its other features, and follows the principles of this presentation.
No need to limit classes to "atomic" features.

Some statistics from EiffelBase

Percentages, rounded.
149 classes, 1823 exported features.
(Includes EiffelLex and EiffelParse, not up to date)

<table>
<thead>
<tr>
<th>Features Range</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 5 features</td>
<td>45</td>
</tr>
<tr>
<td>6 to 10 features</td>
<td>17</td>
</tr>
<tr>
<td>11 to 15 features</td>
<td>11</td>
</tr>
<tr>
<td>16 to 20 features</td>
<td>9</td>
</tr>
<tr>
<td>21 to 40 features</td>
<td>13</td>
</tr>
<tr>
<td>41 to 80 features</td>
<td>4</td>
</tr>
<tr>
<td>81 to 142 features</td>
<td>1</td>
</tr>
</tbody>
</table>

Language and library

The language should be small
The library, in contrast, should provide as many useful facilities as possible.
Key to a non-minimalist library:
- Consistent design.
- Naming.
- Contracts.
Usefulness and power.
The size of feature interfaces

More relevant than class size for assessing complexity.
Statistics from EiffelBase and associated libraries:

<table>
<thead>
<tr>
<th>Feature Types</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of features</td>
<td>1823</td>
</tr>
<tr>
<td>Percentage of queries</td>
<td>59%</td>
</tr>
<tr>
<td>Percentage of commands</td>
<td>41%</td>
</tr>
<tr>
<td>Average number of arguments to a feature</td>
<td>0.4</td>
</tr>
<tr>
<td>Maximum number</td>
<td>3</td>
</tr>
<tr>
<td>No argument</td>
<td>60%</td>
</tr>
<tr>
<td>One argument</td>
<td>37%</td>
</tr>
<tr>
<td>Two arguments</td>
<td>3%</td>
</tr>
<tr>
<td>Three arguments</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

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:
```java
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><code>set_background_color</code></td>
<td><code>background_color</code></td>
</tr>
<tr>
<td>Hidden?</td>
<td>No</td>
<td><code>set_visible</code></td>
<td><code>hidden</code></td>
</tr>
</tbody>
</table>

Naming (classes, features, variables...)

Traditional advice (for ordinary application programming):
- Choose meaningful variable names!
New and old names for EiffelBase classes

<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)
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_erroneous.
This means that making it true is an event worth talking about

Other queries: noun 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

- **Transformations**
  - Conversion
  - Duplication
  - Basic operations

- **Internal**
  - 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`:

```plaintext
enter (i: V; v: T) is
    obsolete
    "Use `put (value, index)'"
    do
        put (v, i)
    end
```

Obsolete classes

```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
```

Complementary material

OOSC2:
- Chapter 22: How to find the classes
- Chapter 23: Principles of class design
End of lecture 13