Object-Oriented Design principles
Designing for reuse

Simplicity
Command / Query Separation principle:
- Clear, understandable interfaces
Systematic naming conventions
Operand / Option Separation:
- Dramatically simplified feature interfaces

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 ...
```

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 complementary aspects:
- Top-down and deductive ("overall design")
- Bottom-up and inductive ("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., 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.)

Beware of "TAXOMANIA"

Active data structures

Original interface for lists:
- `l.insert(i, x)`
- `l.remove()`
- `pos := l.search(x)`
- `l.insert_left_of_value(...)`
- `l.insert_right_of_value(...)`
- `l.insert_by_position(...)`

A typical use:
- `j := l.search(x); l.insert(j + 1, y)`

The revised 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`

A list seen as an active data structure

<table>
<thead>
<tr>
<th>before</th>
<th>item</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>back</td>
<td>forth</td>
<td>count</td>
</tr>
</tbody>
</table>

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.

Changing a question shouldn't change the answer

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, we may substitute 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 \( n := \text{getint}() \)):

\[
\begin{align*}
\text{io.read_integer} \\
\text{n := io.last_integer}
\end{align*}
\]

A typical style

\[
\begin{align*}
\text{your_object.perform_computation} \\
\text{ok := your_object.is成功的} \\
\text{if your_object.is成功 then} \\
\text{value := your_object.results} \end{align*}
\]

Solving \( Ax = b \)

First attempt:

\[
\begin{align*}
\text{if not A.singular then} \\
\text{x := solution (A, b)} \end{align*}
\]

Better:

\[
\begin{align*}
\text{A.try_to_solve (b)} \\
\text{if not A.was_singular then} \\
\text{x := A.solution} \end{align*}
\]

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).

\[
\begin{align*}
\text{APPLICATION} \quad \text{Insert} (x, i + k + 1) \\
\text{LIBRARY} \quad \text{Insert} (x; G; i, \text{INTEGER}) \quad \text{require} \\
\quad i \leq 0 \quad i \equiv \text{count} + 1
\end{align*}
\]

Designing for consistency: An example

Describing active structures properly: can after also be before?

Symmetry:

\[
\begin{array}{c|c}
\text{start} & \text{finish} \\
\text{forth} & \text{back} \\
\text{after} & \text{before}
\end{array}
\]

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

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

Designing for consistency

Typical iteration:

\[
\begin{align*}
\text{from start until after loop} \\
\text{some_action (item)} \\
\text{forth} \end{align*}
\]

Conventions for an empty structure?

- \( after \) must be true for the iteration to stop
- For symmetry: \( before \) should be true too

But this does not work for an empty structure (\( \text{count} = 0 \), see invariant A): should \( \text{index} \) be 0 or 1?
Designing for consistency

To obtain a consistent convention we may transform the invariant into:

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

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

Symmetric but 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):

\[
\begin{align*}
0 \leq \text{index} \\
\text{index} \leq \text{count} + 1 \\
\text{before} &= (\text{index} = 0) \\
\text{after} &= (\text{index} = \text{count} + 1) \\
\text{not (after and before)}
\end{align*}
\]

Valid cursor positions

Non-empty structure

<table>
<thead>
<tr>
<th>before</th>
<th>not after</th>
<th>not before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The case of an empty structure

<table>
<thead>
<tr>
<th>before</th>
<th>not after</th>
<th>not before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can after also be before? Lessons from the example

General principles:

- **Consistency**: "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):

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

How do we measure class size?
- Source lines
- Number of features

How to treat secret features?
- Internal size: includes non-exported features
- External size: includes exported features only

How to deal with 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

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.

Some statistics from EiffelVision 1

Percentages, rounded. 546 classes, 3666 exported features.

Including non-exported features

Percentage rounded. All features (about 7600).

Ratio of total features to exported features: 1.27 (EiffelBase), 1.44 (EiffelVision)
Minimalism?

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 (exported features only):

<table>
<thead>
<tr>
<th>Feature</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>Average number of arguments to a feature</td>
<td>0.6</td>
</tr>
</tbody>
</table>

With non-exported features

Average number of arguments to a feature | 0.5 |
Maximum number | 6
No argument | 57%
One argument | 36%
Two arguments | 5%
Three arguments | 1%
Four arguments | 0.6%
Five or six arguments | 0.2%

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, ...)`

Complementary material

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

End of lecture 16