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 (i)`
- `pos := l.search (x)`
- `l.insert_left_of_value (...)`
- `l.insert_right_of_value (...)`
- `l.insert_by_position (...)`

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 typical use:

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

before

item

after

1

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.

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).
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} \\
n := \text{io.last_integer}
\end{align*}
A typical style

your_object.perform_computation

ok := your_object.is_successful

if your_object.is_successful then
    value := your_object.results
end
Solving $Ax = b$

First attempt:

```java
if not A.singular then
    x := solution (A, b)
end
```

Better:

```java
A.try_to_solve (b)
if not A.was_singular then
    x := A.solution
end
```
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 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:

\[
A \begin{cases} 
  \text{after} = (\text{index} = \text{count} + 1) \\
  \text{before} = (\text{index} = 0) 
\end{cases}
\]
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 to stop
- 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?
Designing for consistency

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

\[
\begin{align*}
& after = (is\_empty \text{ or } (index = count + 1)) \\
& before = (is\_empty \text{ or } (index = 0)) \\
& \quad \text{-- Hence: } is\_empty = (before \text{ and } after)
\end{align*}
\]

Symmetric but leads to frequent tests of the form

\[
\text{if after and not is\_empty then } \ldots
\]

instead of just

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

Invariant (partial):

- $0 \leq index$
- $index \leq count + 1$
- $before = (index = 0)$
- $after = (index = count + 1)$
- $not (after \text{ and } before)$

Valid cursor positions:

- before
- not after
- not after
- $1 \leq index; index \leq count$
- after
- not before
- after
- not before

0 1 item count count + 1
Non-empty structure

\[ \text{before} \quad \text{not after} \]
\[ \text{not after, not before} \quad 1 \leq \text{index}; \text{index} \leq \text{count} \]
\[ \text{after} \]
The case of an empty structure

Valid cursor positions

(before not after)

0

(before not after)

1 (i.e. count + 1)

(after not before)
Can after also be before? Lessons from the 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):

\[
\text{put is require not full}
\]
\[
\text{do ... ensure ... end}
\]
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

Incremental size is most interesting (and 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.

<table>
<thead>
<tr>
<th>Number of Features</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>
Some statistics from EiffelVision 1

Percentages, rounded. 546 classes, 3666 exported features.

<table>
<thead>
<tr>
<th>Features Range</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 5 features</td>
<td>68</td>
</tr>
<tr>
<td>6 to 10 features</td>
<td>12</td>
</tr>
<tr>
<td>11 to 15 features</td>
<td>7</td>
</tr>
<tr>
<td>16 to 20 features</td>
<td>4</td>
</tr>
<tr>
<td>21 to 40 features</td>
<td>6</td>
</tr>
<tr>
<td>41 to 78 features</td>
<td>2</td>
</tr>
</tbody>
</table>
Including non-exported features

Percentage rounded. All features (about 7600).

<table>
<thead>
<tr>
<th>Feature Range</th>
<th>Base</th>
<th>Vision</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 5 features</td>
<td>37</td>
<td>55</td>
</tr>
<tr>
<td>6 to 10 features</td>
<td>23</td>
<td>18</td>
</tr>
<tr>
<td>11 to 15 features</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>16 to 20 features</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>21 to 40 features</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>41 to 80 features</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>81 or more features</td>
<td>2</td>
<td>0.4</td>
</tr>
</tbody>
</table>

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>Value</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>
With non-exported features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Average number of arguments to a feature</th>
<th>Maximum number</th>
</tr>
</thead>
<tbody>
<tr>
<td>No argument</td>
<td>0.5</td>
<td>6</td>
</tr>
<tr>
<td>One argument</td>
<td>57%</td>
<td></td>
</tr>
<tr>
<td>Two arguments</td>
<td>36%</td>
<td></td>
</tr>
<tr>
<td>Three arguments</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>Four arguments</td>
<td>1%</td>
<td></td>
</tr>
<tr>
<td>Five or six arguments</td>
<td>0.6%</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.2%</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) \textit{print} (\texttt{real\_value}, \texttt{number\_of\_significant\_digits}, \texttt{zone\_length}, \texttt{number\_of\_exponent\_digits}, ...)

(O-O) \textit{my\_window}.\textit{display} (\texttt{x\_position}, \texttt{y\_position}, \texttt{height}, \texttt{width}, \texttt{text}, \texttt{title\_bar\_text}, \texttt{color}, ...)

Complementary material

OOSC2:

- Chapter 22: How to find the classes
- Chapter 23: Principles of class design
End of lecture 16