Levels of reusability

0- Used in one system.
1- Used in several systems built by the same person.
2- Used in several systems built by the same group or company.
3- Used in several systems built by people that are in contact with the developers
4- Used by groups unknown to the developers.
Component Quality Model

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension
Component Quality Model

A: Acceptance

A.1 Some reuse attested
A.2 Producer reputation
A.3 Published evaluations

B: Behavior

C: Constraints

D: Design

E: Extension
Component Quality Model

A: Acceptance

B: Behavior

B.1 Examples
B.2 Usage documentation
B.3 Preconditioned
B.4 Some postconditions
B.5 Full postconditions
B.6 Observable invariants

C: Constraints

D: Design

E: Extension
Component Quality Model

A: Acceptance

B: Behavior

C: Constraints

C.1 Platform spec
C.2 Ease of use
C.3 Response time
C.4 Memory occupation
C.5 Bandwidth
C.6 Availability
C.7 Security

D: Design

E: Extension
Contract levels

Type

Functional specification

Performance specification

Quality of Service

(Source: Jézéquel, Mingins et al.)
Component Quality Model

A: Acceptance

B: Behavior

C: Constraints

D: Design

E: Extension

E.1 Portable across platforms
E.2 Mechanisms for addition
E.3 Mechanisms for redefinition
E.4 User action pluggability
Component Quality Model

A: Acceptance

B: Behavior

C: Constraints

D: Design

D.1 Precise dependency doc
D.2 Consistent API rules
D.3 Strict design rules
D.4 Extensive test cases
D.5 Some proved properties
D.6 Proofs of preconditions, postconditions & invariants

E: Extension
The high road: towards proofs?

A: Acceptance

B: Behavior

C: Constraints

D: Design

D.1 Precise dependency doc
D.2 Consistent API rules
D.3 Strict design rules
D.4 Extensive test cases
D.5 Some proved properties
D.6 Proofs of preconditions, postconditions & invariants

E: Extension
The experience of EiffelBase

Designed 1985-1986, revised 1988

About 180 classes covering fundamental structures and algorithms ("Knuthware")

Attempt at "Linnaean" classification of these base structures
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: in 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 are arrays!]
... Set up the non-default properties of \( e \) ...

e.solve

... Answer is now in \( e.x \) and \( e.y \) ...
Our experience: Eiffelbase

Collection classes ("Knuthware")

Consistency principle

Strict design principles: command-query separation, operand-option separation, taxonomy, uniform access...

Strict interface and style rules
Where in an object-oriented hierarchy does the notion of “pseudo-random number” fit?
Pseudo-random numbers

Traditional scheme:

\[ x := \text{random\_start} \left( \text{my\_seed} \right) \]

... loop
  \[ x := \text{random\_next} \left( x \right) \]
  ...

end
The exercise

Where in an object-oriented hierarchy does the notion of “pseudo-random number” fit?
Hints

Hint 1: It’s in the abstraction, stupid
Hints

Hint 1: It’s in the abstraction, stupid

Hint 2: It’s not “random number”
Hints

Hint 1: It’s in the abstraction, stupid

Hint 2: It’s not “random number”

Hint 3: I can’t teach you this
Eiffelbase hierarchy
What makes a good data abstraction?

Good omens:

- Can talk about it in substantive terms
- Several applicable “features”
  - Some are queries, some are commands
    (Ask about instances / Change instances)
- If variant of other, adds or redefines features
  (Beware of taxomania)
- Corresponds to clear concept of one of:
  - **Analysis** (unit of modeling of some part of the world)
  - **Design** (unit of architectural decomposition)
  - **Implementation** (useful data structure)
What makes a good data abstraction?

Bad omens:

“This class does ...”
Name is verb, e.g. “Analyse”
Very similar to other class
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

Key to the construction of a good library is the search for the best abstractions.
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).
Active data structures

OLD INTERFACE FOR LISTS:

l.insert (i,x)
l.remove (i)
pos := l.search(x)
l.insert_by_value (...)
l.insert_by_position (...)
l.search_by_position (...)

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 as an active data structure

before

item

after

back

forth

index

count
Objects as machines
Our exercise

Where in an object-oriented hierarchy does the notion of “pseudo-random number” fit?
Top of Eiffelbase hierarchy
A list as an active data structure

before

item

after

back

forth

index

count
Random number sequence

1

\textit{start (seed)}

item

\textit{index}

\textbf{no “after”}

\textbf{no “count” (?)}

\textbf{forth}
Getting random numbers

Traditional scheme:

\[
x := \text{random\_start} (\text{my\_seed})
\]

... loop

\[
x := \text{random\_next} (x)
\]

...

end
Getting random numbers

Traditional scheme:

\[ x := \text{random\_start} \left( \text{my\_seed} \right) \]
... loop
\[ x := \text{random\_next} \left( x \right) \]
...
end

O-O scheme:

\[ \text{my\_sequence}: \text{RANDOM} \]
... create \text{my\_sequence}.\text{start} \left( \text{my\_seed} \right)
... loop
\[ \text{my\_sequence}.\text{forth} \]
\[ x := \text{my\_sequence}.\text{item} \]
end
Command-query separation principle

Calling a function must not change the target object’s state

This principle excludes many common schemes, such as using functions for input (e.g. C’s `getint` or equivalent).
Feature categories: by role

- Command
- Query
- Feature
- Procedure
- Function
- No result
- Attribute
- Returns result
- Computation
- Memory
Feature categories: by implementation

- Procedure
  - No result
- Routine
  - Returns result
- Computation
- Memory
- Function
- Attribute
Feature categories

- Command
  - No result
- Feature
- Query
  - Returns result
- Function
  - Returns result
  - Memory
- Procedure
  - No result
- Routine
- Computation
  - Memory
- Attribute
  - Memory
Command-query separation principle

Calling a function must not change the target object’s 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, it’s OK to substitute one for the other.

If \( a = b \), then \( f(a) = f(b) \) for any \( f \).

Prohibits functions with side effects.

Also:

For any integer \( i \), normally \( i + i = 2i \)

But even if \( \text{getint}() = 2 \), \( \text{getint}() + \text{getint}() \) is usually not equal to 4.
“... our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.”

Dijkstra, GOTO statement considered harmful, 1968
Command-query separation

Input mechanism (instead of \texttt{n := getint()}):

\begin{verbatim}
io.read_integer
n := io.last_integer
\end{verbatim}
Uniform Access principle

Facilities managed by a module must be accessible to clients in the same way whether implemented by computation or by storage.
\texttt{update\_cartesian} is
\begin{verbatim}
require
  polar\_ok: polar\_uptodate
do
  if not cartesian\_uptodate then
    internal\_x := ro * cos (theta)
    internal\_y := ro * sin (theta)
  end
ensure
  cart\_ok: cartesian\_uptodate
  polar\_ok: polar\_uptodate
end
\end{verbatim}
x: \textit{REAL} is \\
\hspace{1cm} \textit{Abscissa of current point}

\begin{verbatim}
do
   if not cartesian_available then
     update_cartesian
   end

Result := x_internal
\end{verbatim}

\textbf{ensure}

\hspace{1cm} cartesian_ok: cartesian_available

\textbf{end}
Adding two complex numbers

plus (other: COMPLEX ) is
  -- Add other to current complex number.
do
  update_cartesian
  x_internal := x_internal + other.x
  y_internal := y_internal + other.y
ensure
cartesian_ok: cartesian_available
end
Representation invariant

\[
\text{invariant}
\]

\[
\text{cartesian\_uptodate or polar\_uptodate}
\]
Uniform access

$\text{balance} = \text{list\_of\_deposits.total} - \text{list\_of\_withdrawals.total}$

(A1)

list_of_deposits

list_of_withdrawals

balance

(A2)

list_of_deposits

list_of_withdrawals
Uniform Access principle

Facilities managed by a module must be accessible to clients in the same way whether implemented by computation or by storage.
Uniform access through feature call

To access a property of a point \( p \), the Eiffel notation is the same regardless of the representation, e.g.

\[
p.x
\]

applicable in

- cartesian representation (\( x \) is an attribute)
- polar representation (\( x \) is a function)

Field access or computation.

No difference for clients (except speed).
How big should a class be?
How big should a class be?

As big as it needs to – what matters more is consistency of the underlying data abstraction

Example: STRING
EiffelBase statistics

Percentages, rounded.

149 classes, 1823 exported features

<table>
<thead>
<tr>
<th>Feature Range</th>
<th>Percentage</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>

(All statistics here from 1994, but picture hasn’t changed much.)
EiffelVision 1 statistics

Percentages, rounded.

546 classes, 3666 exported features

<table>
<thead>
<tr>
<th>Features</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>

(All statistics here from 1994, but picture hasn’t changed much.)
Minimalism?

Language should be small.
Library should provide as many useful facilities as possible.

Key to a non-minimalist library:

Consistent design
Naming
Contracts

Usefulness and power
## Size of feature interfaces

Statistics from EiffelBase (exported features only):

<table>
<thead>
<tr>
<th>Feature Type</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 arguments</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>
## Size of feature interfaces

Including non-exported features:

<table>
<thead>
<tr>
<th>Average number of arguments to a feature</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number</td>
<td>6</td>
</tr>
<tr>
<td>No arguments</td>
<td>57%</td>
</tr>
<tr>
<td>One argument</td>
<td>36%</td>
</tr>
<tr>
<td>Two arguments</td>
<td>5%</td>
</tr>
<tr>
<td>Three arguments</td>
<td>1%</td>
</tr>
<tr>
<td>Four arguments</td>
<td>0.6%</td>
</tr>
<tr>
<td>Five or six arguments</td>
<td>0.6%</td>
</tr>
</tbody>
</table>
Size of feature interfaces

Statistics from EiffelVision 1 (546 classes, exported only)

<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of features</td>
<td>3666</td>
</tr>
<tr>
<td>Percentage of queries</td>
<td>39%</td>
</tr>
<tr>
<td>Percentage of commands</td>
<td>61%</td>
</tr>
<tr>
<td>Average number of arguments to a feature</td>
<td>0.7</td>
</tr>
<tr>
<td>Maximum number</td>
<td>7</td>
</tr>
<tr>
<td>No argument</td>
<td>49%</td>
</tr>
<tr>
<td>One argument</td>
<td>32%</td>
</tr>
<tr>
<td>Two arguments</td>
<td>15%</td>
</tr>
<tr>
<td>Three arguments</td>
<td>3%</td>
</tr>
<tr>
<td>Four arguments</td>
<td>0.4%</td>
</tr>
<tr>
<td>Five to seven arguments</td>
<td>0.4%</td>
</tr>
</tbody>
</table>
Operands and options

Two possible kinds of arguments to a feature:

Operands: values on which feature will operate
Options: modes that govern how features will operate.

Example: printing a real. The number is an operand; format properties (number of significant digits, width...) are options.

\[
\text{print (real_value, number_of_significant_digits, zone_length, number_of_exponent_digits, ...)}
\]

\[
\text{my_window.display(xposition, yposition, height, width, text, title_bar_text, color, ...)}
\]
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 PRINCIPLE
The arguments of a feature should only be operands

Options should have default values, with procedures to set different values if requested.

For example

```python
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</td>
<td>hidden</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set_hidden</td>
<td></td>
</tr>
</tbody>
</table>
# Naming (1)

<table>
<thead>
<tr>
<th>CLASS</th>
<th>FEATURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>enter</td>
</tr>
<tr>
<td>STACK</td>
<td>push</td>
</tr>
<tr>
<td>QUEUE</td>
<td>add</td>
</tr>
<tr>
<td>HASH_TABLE</td>
<td>insert</td>
</tr>
<tr>
<td>CLASS</td>
<td>FEATURES</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
</tr>
<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)
Contracts
Comments
Some standard names

Queries:
- count
- item
- infix "@"
- to_external, to_c, from_external

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

Boolean queries:
- writable, readable, extendible, prunable
- empty, full
- capacity

-- Array access:
- \( a.item(i) \) or \( a @ i \)

-- Rejected names:
- \( \text{if } s\text{.addable then } s\text{.add}(v) \end \)
- \( \text{if } s\text{.deletable then } s\text{.delete}(v) \end \)

-- Usual invariants:
- empty = (count = 0)
- full = (count = capacity)
Grammatical rules


Boolean queries: adjectives, e.g. full. Also (especially in case of potential ambiguity) names of the form is_some_property. Example: is_first.

➢ In all cases, you should usually choose the form of the property that is false by default at initialization (making it true is an event worth talking about). Example: is_erroneous.

Other queries: nouns or adjectives. Examples: count, error_window.

Do not use verbs for queries, in particular functions; this goes with the command-query separation principle (prohibition of side-effects in functions).
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):

- Initialization
- Access
- Measurement
- Comparison
- Status report
- Status setting
- Cursor movement
- Element change
- Removal
- Resizing
- Transformation
- Conversion
- Duplication
- Basic operations
- Obsolete
- Inapplicable
- Implementation
- Miscellaneous
Obsolete features and classes

A central problem in the computer field: how to reconcile progress with the protection of 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

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
Component development

Component development is Formula-1 programming

Perfectionism Is Good