Designing for reuse
Lecture 13
Designing for reuse

“Formula-1 programming”

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

\[ \text{nonlinear_ode} \]

\[ \text{(equation_count: in INTEGER; epsilon: in out DOUBLE; func: procedure} \]
\[ \quad \text{(eq_count: INTEGER; a: DOUBLE;} \]
\[ \quad \text{eps: DOUBLE; b: ARRAY [DOUBLE];} \]
\[ \quad \text{cm: pointer Libtype);} \]
\[ \quad \text{left_count, coupled_count: INTEGER ...)} \]

[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 ...

\texttt{e.solve}

... Answer available in \texttt{e.x} and \texttt{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

CONTAINER

BOX

FINITE

INFINITE

BOUNDARY

UNBOUNDARY

COUNTABLE

TABLE

ACTIVE

INTEGER_INTERVAL

BILINEAR

ARRAY

STRING

HASH_TABLE

INDEXABLE

CURSOR_STRUCTURE

INDEXABLE

CURSOR_STRUCTURE

DISPENSER

SEQUENCE

STACK

QUEUE

RESIZABLE

FINITE

INFINITE

BOUNDARY

UNBOUNDARY

COUNTABLE

TABLE

ACTIVE

INTEGER_INTERVAL

BILINEAR

ARRAY

STRING

HASH_TABLE
Active data structures

Old interface for lists:

\[
\begin{align*}
&l\text{.insert}(i, x) \\
&l\text{.remove}(i) \\
&\text{pos} := l\text{.search}(x) \\
&l\text{.insert\_by\_value}(\ldots) \\
&l\text{.insert\_by\_position}(\ldots) \\
&l\text{.search\_by\_position}(\ldots)
\end{align*}
\]

New interface:

Queries:

\[
\begin{align*}
l\text{.index} & \quad l\text{.item} \\
l\text{.before} & \quad l\text{.after}
\end{align*}
\]

Commands:

\[
\begin{align*}
l\text{.start} & \quad l\text{.forth} \\
l\text{.search}(x) & \quad l\text{.finish} \\
l\text{.put}(x) & \quad l\text{.back} \\
l\text{.remove} & \quad l\text{.go}(i)
\end{align*}
\]

-- Typical sequence:

\[
\begin{align*}
j & := l\text{.search}(x); \\
l\text{.insert}(j + 1, y)
\end{align*}
\]
A list seen as an active data structure
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 $n := \text{getint}()$):

\[
\begin{align*}
io\text{.read}\_\text{integer} \\
n := \io\text{.last}\_\text{integer}
\end{align*}
\]
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).

**Library**

\[
\text{insert} \ (x: G; \ i: \text{INTEGER})
\]

\[
\begin{align*}
  \text{require} \\
  i & \geq 0 \\
  i & \leq \text{count} + 1
\end{align*}
\]

**Application**

\[
\text{l.insert} \ (x, j + k + 1)
\]
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.

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

Valid cursor positions:

- before
- not before
- not after
- after
Designing for consistency

Typical iteration:

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?
Designing for consistency

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

\[
\begin{align*}
\text{after} &= (\text{is}\_\text{empty} \lor (\text{index} = \text{count} + 1)) \\
\text{before} &= (\text{is}\_\text{empty} \lor (\text{index} = 0))
\end{align*}
\]

\[\text{B} \quad \text{-- Hence: } \text{is}\_\text{empty} = (\text{before} \land \text{after})\]

Symmetric but unpleasant. Leads to frequent tests

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

instead of just

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

Invariant (partial):

\[ 0 \leq \text{index} \leq \text{count} + 1 \]

\[ \text{before} = (\text{index} = 0) \]

\[ \text{after} = (\text{index} = \text{count} + 1) \]

\[ \text{not} (\text{after and before}) \]

Valid cursor positions
The case of an empty structure

Valid cursor positions

0

1 (i.e. \textit{count} + 1)

\textit{before}
\textit{not}
\textit{after}

\textit{after}
\textit{not}
\textit{before}
Can after also be 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):

\[
\begin{align*}
\text{put is} & \\
\text{require} & \\
& \text{not full} \\
\text{do} & \\
& \ldots \\
\text{ensure} & \\
& \ldots \\
\text{end} & 
\end{align*}
\]
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

Most useful measure is incremental size. Easy to measure.
Incremental size

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>Feature 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>Number of features</th>
<th>1823</th>
</tr>
</thead>
<tbody>
<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) \textit{print} \ (\textit{real\_value, number\_of\_significant\_digits, zone\_length, number\_of\_exponent\_digits, ...})

(O-O) \textit{my\_window\_display} \ (\textit{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:

```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 (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>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARRAY</strong></td>
<td>put</td>
<td>entry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>STACK</strong></td>
<td>put</td>
<td>item</td>
<td>remove</td>
<td>top</td>
<td></td>
</tr>
<tr>
<td><strong>QUEUE</strong></td>
<td>put</td>
<td>item</td>
<td>remove</td>
<td>remove</td>
<td>remove_oldest</td>
</tr>
<tr>
<td><strong>HASH_TABLE</strong></td>
<td>put</td>
<td>item</td>
<td>remove</td>
<td>value</td>
<td>delete</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

Boolean queries:
- writable, readable, extendible, pruneable
- is_empty, is_full

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

-- Some rejected names:
if s.addable then
  s.add(v)
end

if s.deletable then
  s.delete(v)
end

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

Procedures (commands): verbs in infinitive form.
   Examples: \textit{make}, \textit{put}, \textit{display}.

Boolean queries: adjectives
   Example: \textit{full} (older convention)
   Now recommended: \textit{is\_full}, \textit{is\_first}.
   \textbf{Convention:} Choose form that should be false by default
   Example: \textit{is\_erroneous}.
   This means that making it true is an event worth talking about

Other queries: nouns or adjectives.
   Examples: \textit{count}, \textit{error\_window}.

Do not use verbs for queries, in particular functions; this goes with
Command-Query Separation Principle
   Example: \textit{next\_item}, not \textit{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):

- Initialization
  - Creation
  - Access
  - Measurement
  - Comparison
  - Status report

Basic queries

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

Basic commands

- Conversion
- Duplication
- Basic operations

Transformations

- Inapplicable
- Implementation
- Miscellaneous

Internal
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:

```java
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
Complementary material

OOSC2:

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