Software Architecture

Prof. Bertrand Meyer, Dr. Michela Pedroni

ETH Zurich, February-May 2010

Lecture 5: Designing for reuse
What exactly is a component?

A component is a program element such that:

- It may be used by other program elements (not just humans, or non-software systems). These elements will be called “clients”

- Its authors need not know about the clients.

- Clients’ authors need only know what the component’s author tells them.
This is a broad view of components

It encompasses patterns and frameworks

Software, especially with object technology, permits “pluggable” components where client programmers can insert their own mechanisms.

Supports component families
Why reuse?

- Faster time to market
- Guaranteed quality
- Ease of maintenance

Consumer view

Producer view

- Standardization of software practices
- Preservation of know-how
Component quality

The key issue in a reuse-oriented software policy

Bad-quality components are a major risk
  Deficiencies scale up, too

High-quality components can transform the state of the software industry
The culture of reuse

From consumer to producer

**Management support is essential, including financial**

The key step: generalization
A reuse policy

The two principal elements:
- Focus on producer side
- Build policy around a library

Library team, funded by Reuse Tax
Library may include both external and internal components
Define and enforce strict admission criteria
Traditional lifecycle model

Separate tools:
- Programming environment
- Analysis & design tools, e.g. UML

Consequences:
- Hard to keep model, implementation, documentation consistent
- Constantly reconciling views
- Inflexible, hard to maintain systems
- Hard to accommodate bouts of late wisdom
- Wastes efforts
- Damages quality
A seamless model

**Seamless development:**
- Single notation, tools, concepts, principles throughout
- Continuous, incremental development
- Keep model, implementation documentation consistent

**Reversibility:** back and forth

Example classes:
- PLANE, ACCOUNT, TRANSACTION...
- STATE, COMMAND...
- HASH_TABLE...
- TEST_DRIVER...
- TABLE...
The cluster model

Permits dynamic reconfiguration

Mix of sequential and concurrent engineering
Levels of reusability

0 - Usable in some program

1 - Usable by programs written by the same author

2 - Usable within a group or company

3 - Usable within a community

4 - Usable by anyone
Nature or nurture?

Two modes:

- Build and distribute libraries of reusable components (business model is not clear)
- Generalize out of program elements

(Basic distinction:

Program element  ---  Software component)
Prepare for reuse. For example:

- Remove built-in limits
- Remove dependencies on specifics of project
- Improve documentation, contracts...
- Abstract
- Extract commonalities and revamp inheritance hierarchy

Few companies have the guts to provide the budget for this
Keys to component development

Substance: Rely on a theory of the application domain

Form: Obsess over consistency

- High-level: design principles
- Low-level: style
Design principles

Object technology: Module $\equiv$ Type

Design by Contract

Command-Query Separation

Uniform Access

Operand-Option Separation

Inheritance for subtyping, reuse, many variants

Bottom-Up Development

Design for reuse and extension

Style matters
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 6 or 7 arguments, of which 2 or 3 arrays!]
The EiffelMath routine

... Create e and set-up its values (other than defaults) ...

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
What makes a good data abstraction?

Good signs:

- 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)
“Design smells”

Signs that a proposed class may not be right

- “This class does ...”
- Name is verb, e.g. “Analyze”
- 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
  Many classes associated with design patterns fall into this category
- **Implementation classes** - examples: ARRAY, LINKED_LIST

Key to the construction of a good library is the search for the best abstractions
The key to building a library

Devising a theory of the underlying domain
Eiffelbase hierarchy

Representation

CONTAINER

COLLECTION

Access

CONTAINER

COLLECTION

TRAVERSABLE

Iteration

CONTAINER

COLLECTION

Access

CONTAINER

COLLECTION

TRAVERSABLE

Finite

Infinite

Bounded

Unbounded

Countable

Table

Active

Subset

Bilinear

Fixed

Resizable

Indexable

Cursor

Structure

Dispenser

Sequence

Countable
Active data structures

Old interface for lists:

- \( l.\text{insert}(i, x) \)
- \( l.\text{remove}(i) \)
- \( pos := l.\text{search}(x) \)
- \( l.\text{insert}_\text{by}_\text{value}(...) \)
- \( l.\text{insert}_\text{by}_\text{position}(...) \)
- \( l.\text{search}_\text{by}_\text{position}(...) \)

New interface:

Queries:

- \( l.\text{index} \)
- \( l.\text{item} \)
- \( l.\text{before} \)
- \( l.\text{after} \)

Commands:

- \( l.\text{start} \)
- \( l.\text{forth} \)
- \( l.\text{finish} \)
- \( l.\text{back} \)
- \( l.\text{go}(i) \)
- \( l.\text{search}(x) \)
- \( l.\text{put}(x) \)
- \( l.\text{remove} \)

-- Typical use: 
\[ j := l.\text{search}(x); l.\text{insert}(j+1, y) \]
A list seen as an active data structure
Beyond internal cursors

Internal cursors, as in the preceding example, have disadvantages:

- Poorly adapted to recursive routines and concurrency
- Programmers need to remember to reset cursor, e.g.

```plaintext
backup := l.index
from start until after loop
    some_operation (l.item)
    l.forth
end
l.go_i_th (backup)
```
External cursor

The cursor becomes an object:

Operations on a cursor $c$:
- $c$.start
- $c$.forth
- $c$.index
- $c$.item
- $c$.after

$1$ $\text{"Zurich"}$ count

and other commands

and other queries
Loop construct with built-in cursor

Instead of

    local
        c: CURSOR [...] 
    ...
create c.make (my_list)
from c.start until c.after loop
    some_operation (c.item)
    c.forth
end

just use (EiffelStudio 6.5):

    across my_list as c loop some_operation (c.item) end

Structure’s class must be a descendant of ITERABLE. This is the case (6.6) with lists, arrays, hash tables, ...
"across" loop for predicates

```
across my_integer_list as c all c.item > 0 end

across my_integer_list as c some c.item > 0 end
```
Uniform access

Uniform Access principle

It does not matter to the client whether you look up or compute
Uniform access

\[ \text{balance} = \text{list\_of\_deposits}.\text{total} - \text{list\_of\_withdrawals}.\text{total} \]
A self-adapting complex number class

class COMPLEX feature {NONE}
    x_internal, y_internal, ro_internal, theta_internal: REAL
    cartesian_available, polar_available: BOOLEAN

update_cartesian
    require
        polar_ok: polar_available
    do
        if not cartesian_available then
            internal_x := ro \times cos(theta)
            internal_y := ro \times sin(theta)
            cartesian_available := True
        end
    end
ensure
    cart_ok: cartesian_available
    polar_ok: polar_available
end
Representation invariant

\texttt{invariant \hspace{1em} cartesian\_available \textbf{or} polar\_available}
feature
  \texttt{x: REAL}
  \hspace{1cm} -- Abscissa of current point
  do
    \texttt{update\_cartesian}
    \texttt{Result := x\_internal}
  ensure
    \texttt{cartesian\_ok: cartesian\_available}
end
Adding two complex numbers

```plaintext
plus(other: COMPLEX)
   -- 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
```
Commands and queries

**Command-Query Separation principle**

A query must not change the target object’s state.
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`)
Feature classification (reminder)

Client view (specification)

- Command
  - No result
  - Returns result

Feature

Query

Internal view (implementation)

- Routine
  - Computation
  - Memory

Feature

- Procedure
  - Function
  - Attribute

- Memory
  - Computation
  - Memory
Asking a question should not change the answer!
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 using EiffelBase

(instead of $n := \text{getint} ()$):

\[
\begin{align*}
io \text{. read\_integer} \\
n & := \io \text{. last\_integer}
\end{align*}
\]
Include appropriate contracts:

- Contracts help design the libraries right.
- Preconditions help find errors in client software.
- Library documentation fundamentally relies on contracts (interface views).

```
APPLICATION

library

insert(x, j + k + 1)

require

i >= 0
i <= count + 1

LIBRARY

insert(x: G, i: INTEGER)
```
Designing for consistency: An example

Describing active structures properly: can after also be before?

Symmetry:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>finish</td>
</tr>
<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.

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

before

item
"Zurich"

forth

back

finish

start

index

count

after

Cursor

1

count
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:

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

Symmetric but unpleasant. Leads to frequent tests

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

instead of just

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

Invariant (partial):

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

Valid cursor positions

\[
\begin{array}{cccccc}
0 & 1 & \text{item} & \text{count} & \text{count + 1} \\
\text{not after} & \text{not before} & \text{not after} & \text{not before} & \text{not after} & \text{not before}
\end{array}
\]
The case of an empty structure

0

before

not

after

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

after

not

before

Valid cursor positions
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):

\[\text{put} \]
\[\text{require} \]
\[\text{not full} \]
\[\text{do} \]
\[\ldots \]
\[\text{ensure} \]
\[\ldots \]
\[\text{end} \]
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.
Feature classification (reminder)

Client view (specification)
- Command
- Feature
- Query
- No result
- Returns result

Internal view (implementation)
- Routine
- Feature
- Function
- Attribute
- Computation
- Memory
Another classification

Feature of a class
- Inherited
- From parent
- New in class

Immediate
- Redeclared
- Changed
- Kept
- Unchanged

Incremental size
- Redefined
- Effected
- Was deferred
- Had an implementation
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.
How big should a class be?

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

Example: *STRING_8*
- 154 immediate features
- 2675 lines of code
EiffelBase statistics

Percentages, rounded.
250 classes, 4408 exported features

<table>
<thead>
<tr>
<th>Features Range</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 5 features</td>
<td>43</td>
</tr>
<tr>
<td>6 to 10 features</td>
<td>14</td>
</tr>
<tr>
<td>11 to 15 features</td>
<td>10</td>
</tr>
<tr>
<td>16 to 20 features</td>
<td>4</td>
</tr>
<tr>
<td>21 to 40 features</td>
<td>17</td>
</tr>
<tr>
<td>41 to 80 features</td>
<td>9</td>
</tr>
<tr>
<td>81 to 142 features</td>
<td>2</td>
</tr>
</tbody>
</table>

(All measures from version 6.0, courtesy Yi Wei)
Percentages, rounded.
733 classes, 5872 exported features

<table>
<thead>
<tr>
<th>Features Range</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 5 features</td>
<td>64</td>
</tr>
<tr>
<td>6 to 10 features</td>
<td>14</td>
</tr>
<tr>
<td>11 to 15 features</td>
<td>8</td>
</tr>
<tr>
<td>16 to 20 features</td>
<td>5</td>
</tr>
<tr>
<td>21 to 40 features</td>
<td>7</td>
</tr>
<tr>
<td>41 to 80 features</td>
<td>2</td>
</tr>
</tbody>
</table>
EiffelVision on Linux

Percentages, rounded.
698 classes, 8614 exported features

<table>
<thead>
<tr>
<th>Features Range</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 5 features</td>
<td>63</td>
</tr>
<tr>
<td>6 to 10 features</td>
<td>13</td>
</tr>
<tr>
<td>11 to 15 features</td>
<td>8</td>
</tr>
<tr>
<td>16 to 20 features</td>
<td>5</td>
</tr>
<tr>
<td>21 to 40 features</td>
<td>8</td>
</tr>
<tr>
<td>41 to 80 features</td>
<td>2</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>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of features</td>
<td>4408</td>
</tr>
<tr>
<td>Percentage of queries</td>
<td>66%</td>
</tr>
<tr>
<td>Percentage of commands</td>
<td>34%</td>
</tr>
<tr>
<td>Average number of arguments to a feature</td>
<td>0.5</td>
</tr>
<tr>
<td>Maximum number</td>
<td>5</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>6%</td>
</tr>
<tr>
<td>Three or more arguments</td>
<td>1%</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>0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number</td>
<td>12</td>
</tr>
<tr>
<td>No arguments</td>
<td>55%</td>
</tr>
<tr>
<td>One argument</td>
<td>36%</td>
</tr>
<tr>
<td>Two arguments</td>
<td>7%</td>
</tr>
<tr>
<td>Three arguments</td>
<td>2%</td>
</tr>
<tr>
<td>Four arguments</td>
<td>0.4%</td>
</tr>
<tr>
<td>Five or six arguments</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
## Size of feature interfaces

**EiffelVision on Windows (733 classes, exported only)**

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of features</td>
<td>5872</td>
</tr>
<tr>
<td>Percentage of queries</td>
<td>56%</td>
</tr>
<tr>
<td>Percentage of commands</td>
<td>44%</td>
</tr>
<tr>
<td>Average number of arguments to a feature</td>
<td>0.5</td>
</tr>
<tr>
<td>Maximum number</td>
<td>10</td>
</tr>
<tr>
<td>No argument</td>
<td>67%</td>
</tr>
<tr>
<td>One argument</td>
<td>23%</td>
</tr>
<tr>
<td>Two arguments</td>
<td>6%</td>
</tr>
<tr>
<td>Three arguments</td>
<td>1.5%</td>
</tr>
<tr>
<td>Four arguments</td>
<td>1.5%</td>
</tr>
<tr>
<td>Five to seven arguments</td>
<td>0.6%</td>
</tr>
</tbody>
</table>
## Size of feature interfaces

**EiffelVision on Linux (698 classes, exported only)**

<table>
<thead>
<tr>
<th>Feature Description</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of features</td>
<td>8614</td>
</tr>
<tr>
<td>Percentage of queries</td>
<td>56%</td>
</tr>
<tr>
<td>Percentage of commands</td>
<td>44%</td>
</tr>
<tr>
<td>Average number of arguments to a feature</td>
<td>0.96</td>
</tr>
<tr>
<td>Maximum number</td>
<td>14</td>
</tr>
<tr>
<td>No argument</td>
<td>49%</td>
</tr>
<tr>
<td>One argument</td>
<td>28%</td>
</tr>
<tr>
<td>Two arguments</td>
<td>15%</td>
</tr>
<tr>
<td>Three arguments</td>
<td>4%</td>
</tr>
<tr>
<td>Four arguments</td>
<td>2%</td>
</tr>
<tr>
<td>Five to seven arguments</td>
<td>1%</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:

```plaintext
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> <code>set_hidden</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><strong>ARRAY</strong></td>
<td><code>put</code> <code>item</code> <code>enter</code> <code>entry</code></td>
</tr>
<tr>
<td><strong>STACK</strong></td>
<td><code>put</code> <code>item</code> <code>push</code> <code>top</code> <code>remove</code></td>
</tr>
<tr>
<td><strong>QUEUE</strong></td>
<td><code>put</code> <code>add</code> <code>item</code> <code>oldest</code> <code>remove</code> <code>remove_oldest</code></td>
</tr>
<tr>
<td><strong>HASH_TABLE</strong></td>
<td><code>put</code> <code>insert</code> <code>item</code> <code>value</code> <code>remove</code> <code>delete</code></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 & result)
- **Assertions**
- **Comments**
Some standard names

Queries (non-boolean):
- count, capacity
- item
to_external, from_external

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

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

-- For creation

-- Usual invariants:
- $0 \leq \text{count} \leq \text{capacity}$
- is_empty = (count = 0)
- is_full = (count = capacity)

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

- if $s$.deletable then
  - $s$.delete(v)
end
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: nouns 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):

- Initialization
- Access
- Measurement
- Comparison
- Status report
- Status setting
- Cursor movement
- Element change
- Removal
- Resizing
- Transformation

Basic queries

- Conversion
- Duplication
- Basic operations

Basic commands

- Inapplicable
- Implementation
- Miscellaneous

Transformations

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:**

\[
\text{enter}(i: V; v: T) \\
\text{obsolete} \\
\text{do} \\
\quad \text{put}(v, i) \\
\text{end}
\]

"Use `put (value, index)`"
class
  \textit{ARRAY\_LIST \{G\}}

\textbf{obsolete}
  
  
  \begin{verbatim}
  \[
  \[
  \text{Use } \textit{MULTI\_ARRAY\_LIST} \text{ instead (same semantics, but new name ensures more consistent terminology).}

  \textit{Caution: do not confuse with } \textit{ARRAYED\_LIST} \text{ (lists implemented by one array each).}
  \]
  \end{verbatim}

\textbf{inherit}
  \textit{MULTI\_ARRAY\_LIST \{G\}}

end
Summary

- Reuse-based development holds the key to substantial progress in software engineering.
- Reuse is a culture, and requires management commitment ("buy in").
- The process model can support reuse.
- Generalization turns program elements into software components.
- A good reusable library proceeds from systematic design principles and an obsession with consistency.
Complementary material

OOSC2:

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