It’s the technology, stupid!
Six easy pieces

Bertrand Meyer

LASER, Biodola, September 2006
Contracts and tests
The cluster model
Dealing with events
Dealing with Void
Open-sourcing EiffelStudio
Towards an O-O process
Software engineering

The collection of techniques — including theories, methods, processes, tools and languages — for developing and operating production software meeting defined standards of quality.
External quality factors

CORRECTNESS
ROBUSTNESS
INTEGRITY

EASE OF USE
REUSABILITY
EXTENDIBILITY
PORTABILITY
EFFICIENCY

...  

Correctness:
- The ability of a software system to perform according to specification, in cases defined by the specification.

Robustness:
- The ability of a software system to react in a reasonable manner to cases not covered by the specification.
Software quality factors

Product quality (immediate):
Correctness
Robustness
Integrity
Ease of use
Ease of learning

Process quality:
Timeliness
Cost-effectiveness

Product quality (long term):
Extendibility
Reusability
Portability

...
The Software Engineering problem

Developing software systems that are

- On time and within budget
- Of high immediate quality
- Possibly large and complex
- Extendible
Models and standards

Capability Maturity Model (CMM)

Characterization of maturity of the software development model of a company
Five levels
Popular with defense contractors, outsourcing companies

Also: ISO 900x quality standards (International Standards Organization)
Welcome to the world of standards

http://www.software.org/quagmire
The two CMMI stages

Staged

ML5
ML4
ML3
ML2
ML 1

...for an established set of process areas across an organization

Continuous

Process Area Capability

0 1 2 3 4 5

PA PA PA

...for a single process area or a set of process areas (PA)
The plan for performing the organizational process focus process, which is often called “the process-improvement plan,” differs from the process action plans described in specific practices in this process area. The plan called for in this generic practice addresses the comprehensive planning for all of the specific practices in this process area, from the establishment of organizational process needs all the way through to the incorporation of process-related experiences into the organizational process assets.
# CMMI levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Process Characteristics</th>
<th>Management Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizing</td>
<td>Focus is on continuous quantitative improvement</td>
<td>In → Out</td>
</tr>
<tr>
<td>Quantitatively Managed</td>
<td>Process is measured and controlled</td>
<td>In → Out</td>
</tr>
<tr>
<td>Defined</td>
<td>Process is characterized for the organization and is proactive</td>
<td>In → Out</td>
</tr>
<tr>
<td>Managed</td>
<td>Process is characterized for projects and is often reactive</td>
<td>In → Out</td>
</tr>
<tr>
<td>Initial</td>
<td>Process is unpredictable, poorly controlled, and reactive</td>
<td>In → Out</td>
</tr>
</tbody>
</table>
Specific and generic goals & practices
## Generic goals and practices

<table>
<thead>
<tr>
<th>Level</th>
<th>Generic Goals</th>
<th>Generic Practices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Achieve Specific Goals</td>
<td>GP 1.1 Perform Base Practices</td>
</tr>
<tr>
<td>2</td>
<td>Institutionalize a Managed Process</td>
<td>GP 2.1 Establish an Organizational Policy, GP 2.2 Plan the Process, GP 2.3 Provide Resources, GP 2.4 Assign Responsibility, GP 2.5 Train People, GP 2.6 Manage Configurations, GP 2.7 Identify and Involve Relevant Stakeholders, GP 2.8 Monitor and Control the Process, GP 2.9 Objectively Evaluate Adherence, GP 2.10 Review Status with Higher Level Mgmt</td>
</tr>
<tr>
<td>3</td>
<td>Institutionalize a Defined Process</td>
<td>GP 3.1 Establish a Defined Process, GP 3.2 Collect Improvement Information</td>
</tr>
<tr>
<td>4</td>
<td>Institutionalize a Quantitatively Managed Process</td>
<td></td>
</tr>
</tbody>
</table>

**Software Engineering**

**ETH**
Institutionalization involves implementing practices that

- Ensure the process areas are effective, repeatable and long lasting
- Provide needed infrastructure support
- Ensure processes are defined, documented, understood
- Enable organizational learning to improve the processes
Establish and Maintain

- This phrase connotes a meaning beyond the component terms; it includes documentation and usage.

Work product

- The term “work product” is used throughout the CMMI Product Suite to mean any artifact produced by a process. These artifacts can include files, documents, parts of the product, services, processes, specifications, and invoices.

Planned Process

- A process that is documented both by a description and a plan. The description and plan should be coordinated, and the plan should include standards, requirements, objectives, resources, assignments, etc.
Model Terminology -3

Performed Process (Capability Level 1)

- A process that accomplishes the needed work to produce identified output work products using identified input work products. The specific goals of the process area are satisfied.

Managed Process (Capability Level 2)

- A “managed process” is a performed process that is planned and executed in accordance with policy; employs skilled people having adequate resources to produce controlled outputs; involves relevant stakeholders; is monitored, controlled, and reviewed; and is evaluated for adherence to its process description.

Defined Process (Capability Level 3)

- A “defined process” is a managed process that is tailored from the organization’s set of standard processes according to the organization’s tailoring guidelines; has a maintained process description; and contributes work products, measures, and other process-improvement information to the organizational process assets.
Generic Practices Summary

The Generic Practices support institutionalization of critical practices for an organization to have a successful process improvement initiative

- Processes will be **executed and managed consistently**
- Processes will **survive staff changes**
- Process **improvement** will be **related to business goals**
- The organization will **not** find itself continuously “reinventing the wheel”
- There will be the commitment to provide **resources** or infrastructure to support or improve the processes
- There will be historical basis for cost **estimation**
The anti-process movement

“eXtreme Programming” (XP), “Agile” methods

Test-driven development
Recommended practices, e.g. Pair Programming
Short iteration cycles

“The revenge of the cubicles”
Lifecycle models

Origin: Royce, 1970, Waterfall model

Scope: describe the set of processes involved in the production of software systems, and their sequencing

“Model” in two meanings of the term:
- Idealized description of reality
- Ideal to be followed
The waterfall model of the lifecycle

1. Feasibility study
2. Requirements
3. Specification
4. Global design
5. Detailed design
6. Implementation
7. V & V
8. Distribution
V-shaped

FEASIBILITY STUDY

REQUIREMENTS ANALYSIS

GLOBAL DESIGN

DETAILED DESIGN

IMPLEMENTATION

UNIT VALIDATION

SUBSYSTEM VALIDATION

SYSTEM VALIDATION

DISTRIBUTION
Arguments for the waterfall

(After B.W. Boehm: *Software engineering economics*)

- The activities are necessary
  - (But: merging of middle activities)

- The order is the right one.
The waterfall model

- Feasibility study
- Requirements
- Specification
- Global design
- Detailed design
- Implementation
- V & V
- Distribution
Problems with the waterfall

Late appearance of actual code.
Lack of support for requirements change — and more generally for extendibility and reusability.
Lack of support for the maintenance activity (70% of software costs?).
Division of labor hampering Total Quality Management.
Impedance mismatches.
Highly synchronous model.
Quality control?

- Analysts
- Designers
- Implementers
- Testers
- Customers
Impedance mismatches

1. As Management requested it.
2. As the Project Leader defined it.
3. As Systems designed it.
4. As Programming developed it.
5. As Operations installed it.
6. What the user wanted.

(Pre-1970 cartoon; origin unknown)
The Spiral model (Boehm)

Figure from: Ghezzi, Jazayeri, Mandrioli, *Software Engineering, 2nd edition*, Prentice Hall
The Spiral model

M.C Escher: Waterval
Tasks

- Analysts
- Designers
- Implementers
- Testers
Seamless development (as in Eiffel)

**Seamless development:**
- Single notation, tools, concepts, principles throughout
- Eiffel is as much for analysis & design as implementation & maintenance
- Continuous, incremental development
- Keep model, implementation and documentation consistent

**Reversibility:** go back and forth
- Saves money: invest in single set of tools
- Boosts quality

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

Mix of sequential and concurrent engineering

Permits dynamic reconfiguration
Analysis classes

defered class SUMMER_SCHOOL_SESSION
inherit EVENT

feature
  series: SUMMER_SCHOOL
  location: EVENT_LOCATION
  dates: TUPLE [DATE, DATE]
  count: INTEGER
  is_scheduled: BOOLEAN

  schedule (d: TUPLE [DATE, DATE]; l: EVENT_LOCATION)
  -- Set date and place to given values.
    require
      location.available (dates)
    deferred
      organizers.available (dates)
    ensure
      dates ~ d
      location ~ l
      is_scheduled

... [Other features] ...

invariant
  web_page.is_released implies is_scheduled
end
Seamless development

Use consistent notation from analysis to design, implementation and maintenance.

Advantages:
- Smooth process. Avoids gaps (improves productivity, reliability).
- Direct mapping from problem to solution, i.e. from software system to external model.
- Better responsiveness to customer requests.
- Consistency, ease of communication.
- Better interaction between users, managers and developers.
Single model principle

Use a single base for everything: analysis, design, implementation, documentation...

Use tools to extract the appropriate views.
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
Levels of reusability for a software element

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)
Generalization

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
Two keys to component development success

**Substance:** Rely on a theory of the application domain

**Form:** Obsess over consistency
- High-level: design principles
- Low-level: style
<table>
<thead>
<tr>
<th>Class</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>put</td>
</tr>
<tr>
<td></td>
<td>enter</td>
</tr>
<tr>
<td></td>
<td>item</td>
</tr>
<tr>
<td></td>
<td>entry</td>
</tr>
<tr>
<td>STACK</td>
<td>put</td>
</tr>
<tr>
<td></td>
<td>push</td>
</tr>
<tr>
<td></td>
<td>item</td>
</tr>
<tr>
<td></td>
<td>top</td>
</tr>
<tr>
<td>QUEUE</td>
<td>put</td>
</tr>
<tr>
<td></td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>item</td>
</tr>
<tr>
<td></td>
<td>oldest</td>
</tr>
<tr>
<td></td>
<td>remove</td>
</tr>
<tr>
<td></td>
<td>remove_oldest</td>
</tr>
<tr>
<td>HASH_TABLE</td>
<td>put</td>
</tr>
<tr>
<td></td>
<td>insert</td>
</tr>
<tr>
<td></td>
<td>item</td>
</tr>
<tr>
<td></td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>remove</td>
</tr>
<tr>
<td></td>
<td>delete</td>
</tr>
</tbody>
</table>
Design principles in the Eiffel method

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
Typical API in a traditional library (NAG)

```java
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 approach

e: ORDINARY_DIFFERENTIAL_EQUATION

create e.make("...values ...")

e.solve

-- Answer available in e.x and e.status ...
Style rules

No routine without header comments

Preconditions always fully expressed

Postconditions and invariants: the more the better

Redundancy OK in class invariants (axioms and theorems)

Standardized layout

Queries never use verbs!

Systematic naming conventions

No exceptions; rules strictly enforced

Class ACCOUNT:

balance, not get_balance
Feature categories: keeping a class in order

class

C

inherit

...

feature -- Category 1

... Feature declarations ...

feature -- Category 2

... Feature declarations ...

feature -- Category \( n \)

... Feature declarations ...

invariant

...

end
Feature categories

Standard categories (the only ones in EiffelBase):

- **Initialization**
- **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**
Simplicity

Command-query separation principle:
   ➢ Clear, understandable interfaces

Systematic naming conventions

Operand-option separation:
   ➢ Dramatically simplified feature interfaces
The view from a traditional library (NAG)

nonlinear_code
(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 components:

- Top-down and deductive (the overall design).
- Bottom-up and inductive (the 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.) TAXOMANIA

Names that warrant some attention: “er” names, e.g. ANALYZER.
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 seen as an active data structure

- before
- item
- after
- 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.

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

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

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

\[
\quad \text{insert} (x: G; i: \text{INTEGER}) \\
\quad \quad \quad \quad \quad \text{require} \\
\quad \quad \quad \quad \quad \quad \quad i >= 0 \\
\quad \quad \quad \quad \quad \quad \quad i <= \text{count} + 1
\]
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>

Valid cursor positions

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

\[
\begin{align*}
\text{after} &= (\text{index} = \text{count} + 1) \\
\text{before} &= (\text{index} = 0)
\end{align*}
\]
Designing for consistency (cont’d)

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 (cont’d)

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

\[
\begin{aligned}
\text{after} &= \text{is\_empty} \text{ or } (\text{index} = \text{count} + 1) \\
\text{before} &= \text{is\_empty} \text{ or } (\text{index} = 0)
\end{aligned}
\]

-- Hence: \text{is\_empty} = (\text{before and after})

Symmetric but unpleasant. Leads to frequent tests of the form

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

instead of just

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

Invariant (partial):
\[ 0 \leq \text{index} \]
\[ \text{index} \leq \text{count} + 1 \]
\[ \text{before} = (\text{index} = 0) \]
\[ \text{after} = (\text{index} = \text{count} + 1) \]
\[ \text{not (after and before)} \]

Valid cursor positions
The case of an empty structure

Valid cursor positions

before not after

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

after not before
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):

\[
\text{put is require not full do } \ldots \text{ ensure } \ldots \text{ end}
\]
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 Lex and Parse.)

<table>
<thead>
<tr>
<th>Features 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>
Some statistics from EiffelVision

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>Features 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 revisited

The language should be small (ETL: “The language design should provide a good way to express every operation of interest; it should avoid providing two.”)

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 Base, Lex and Parse (exported features only):

<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>
Including non-exported features:

<table>
<thead>
<tr>
<th>Number of Arguments</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>No argument</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>0.2%</td>
</tr>
</tbody>
</table>
The size of feature interfaces (cont’d)

Statistics from EiffelVision (546 classes, exported features only):

<table>
<thead>
<tr>
<th>Feature Description</th>
<th>Percentage</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 arguments</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>
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. The number is an operand; format properties (e.g. number of significant digits, width) are options.

```plaintext
print(real_value, number_of_significant_digits,
      zone_length, number_of_exponent_digits, ...)

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.
Operands and options

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:

```plaintext
my_window.set_background_color("dd-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>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, entry</td>
</tr>
<tr>
<td>STACK</td>
<td>push, top, pop</td>
</tr>
<tr>
<td>QUEUE</td>
<td>add, oldest, remove_oldest</td>
</tr>
<tr>
<td>HASH_TABLE</td>
<td>insert, value, delete</td>
</tr>
</tbody>
</table>
### Naming (2)

<table>
<thead>
<tr>
<th>Class</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<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).
- Assertions.
- 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:
- `if s.addable then`
  - `s.add (v)`
- `end`
- `if s.deletable then`
  - `s.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).
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 (cont’d)

**Standard categories (the only ones in EiffelBase):**

<table>
<thead>
<tr>
<th>Yellow Section</th>
<th>Blue Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>Conversion</td>
</tr>
<tr>
<td>Access</td>
<td>Duplication</td>
</tr>
<tr>
<td>Measurement</td>
<td>Basic operations</td>
</tr>
<tr>
<td>Comparison</td>
<td>Obsolete</td>
</tr>
<tr>
<td>Status report</td>
<td>Inapplicable</td>
</tr>
<tr>
<td>Status setting</td>
<td>Implementation</td>
</tr>
<tr>
<td>Cursor movement</td>
<td>Miscellaneous</td>
</tr>
<tr>
<td>Element change</td>
<td></td>
</tr>
<tr>
<td>Removal</td>
<td></td>
</tr>
<tr>
<td>Resizing</td>
<td></td>
</tr>
<tr>
<td>Transformation</td>
<td></td>
</tr>
</tbody>
</table>
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 \texttt{ARRAY}:

\begin{verbatim}
enter (i: V; v: T) is
  obsolete "Use `put (value, index)`"
  do
    put (v, i)
  end
\end{verbatim}
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\[G\]

deprecated
Cluster development

Bottom-up development: from the most general clusters (providing utility functions) to the most application-specific ones.

Flexible scheduling of clusters - depending on resources, team experience, customer and management demands. Waterfall is one extreme; “trickle” is the other.

Sub-lifecycle sequencing: specification, design and implementation, validation, generalization.

Relations between clusters: each cluster may be a client of lower-level ones.
Quality goals: the Osmond curves

Other qualities

Common

Functionality

Desirable

Envisaged

Release