Chair of Software Engineering

Software Architecture

Bertrand Meyer, Carlo A. Furia, Martin Nordio

ETH Zurich, February-May 2011

Lecture 6: 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?

Consumer view

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

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

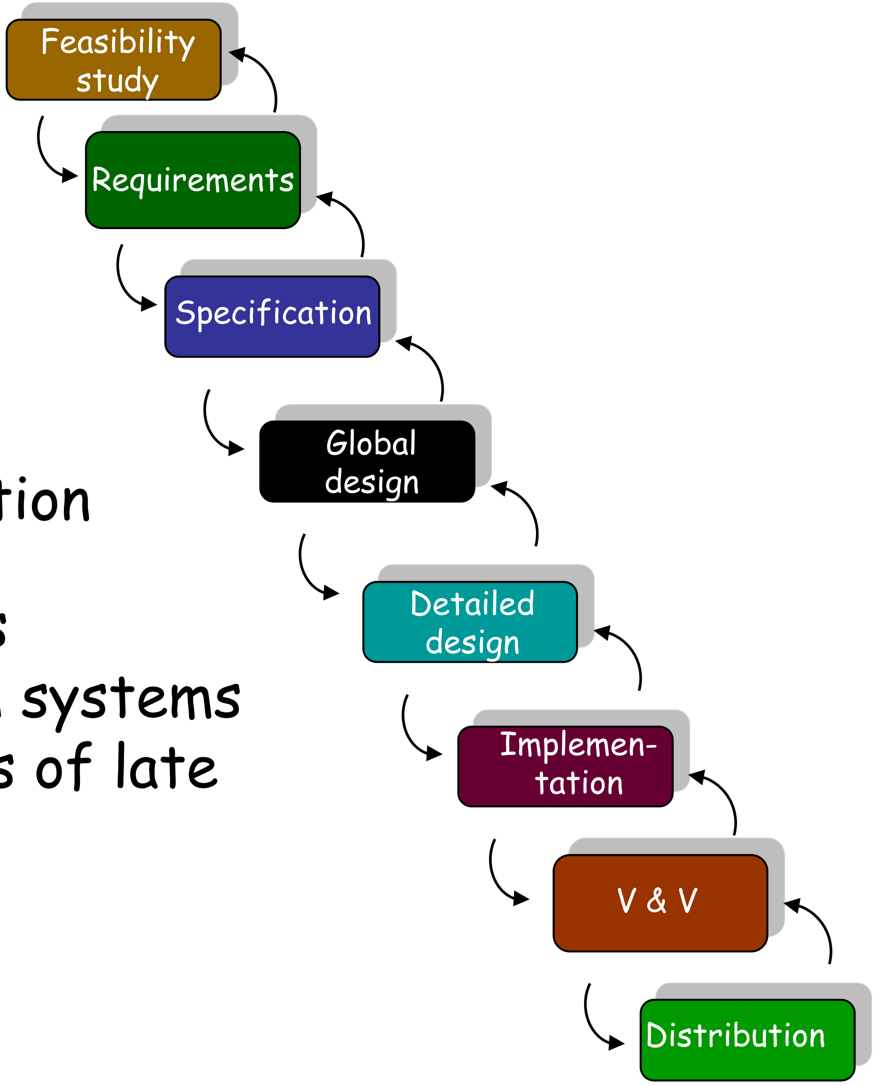
Traditional lifecycle model

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



Feasibility study

Requirements

Specification

Global design

Detailed design

Implementation

V & V

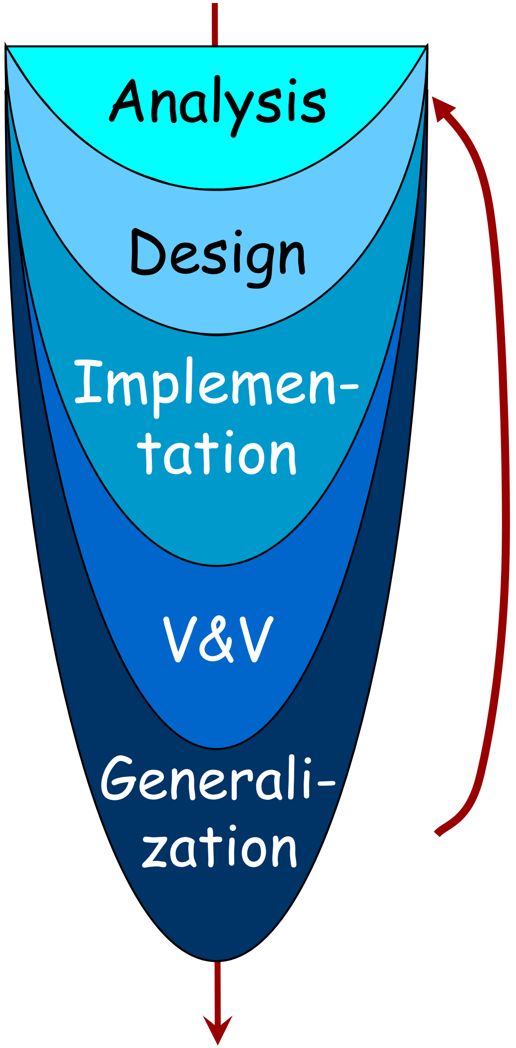
Distribution

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…

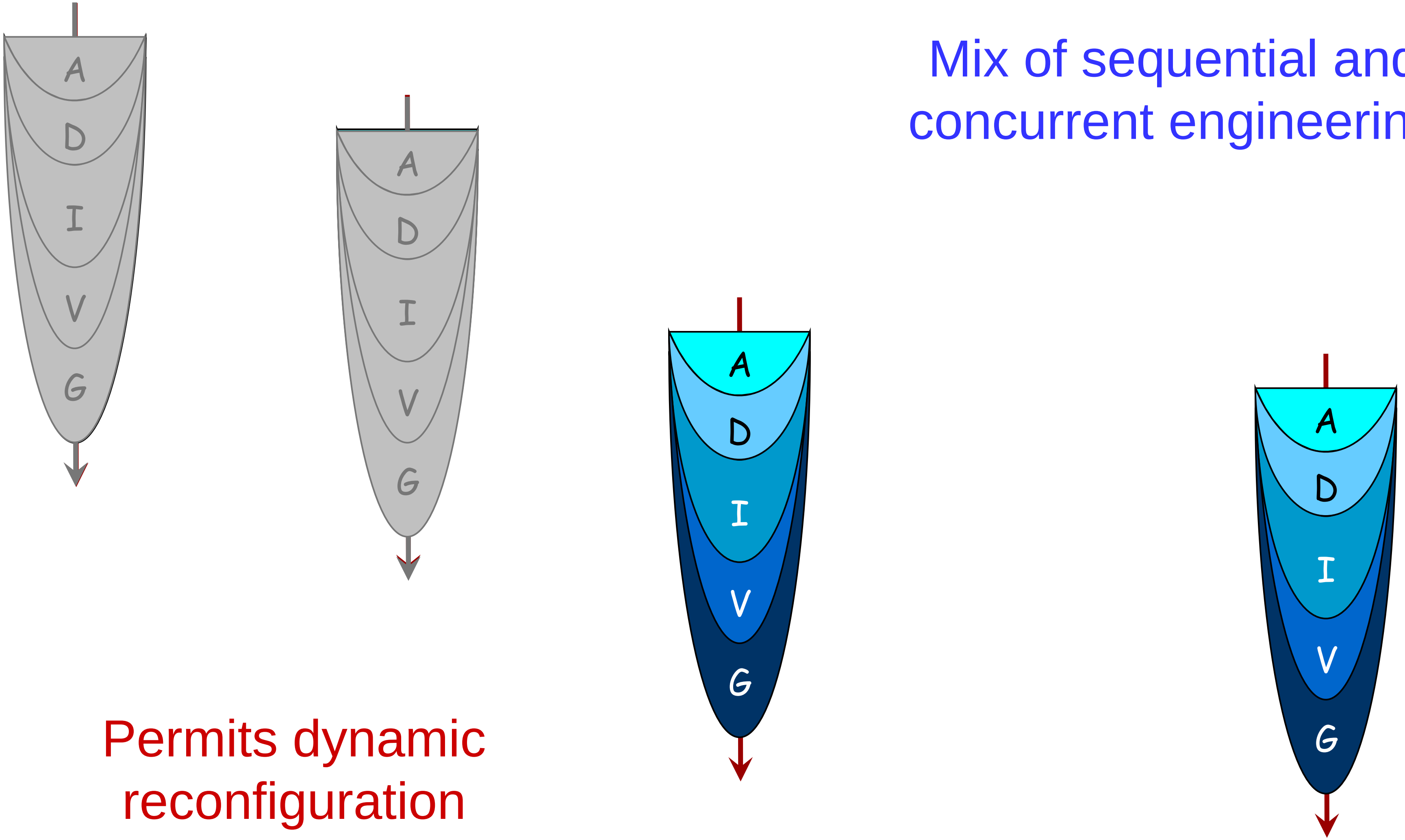
STATE, COMMAND…

HASH_TABLE…

TEST_DRIVER…

TABLE…

The cluster model



Mix of sequential and concurrent engineering

Permits dynamic reconfiguration

Levels of reusability

0 - Usable in some program

1 - Usable by programs written by the same author

2 - Usable within a group or company

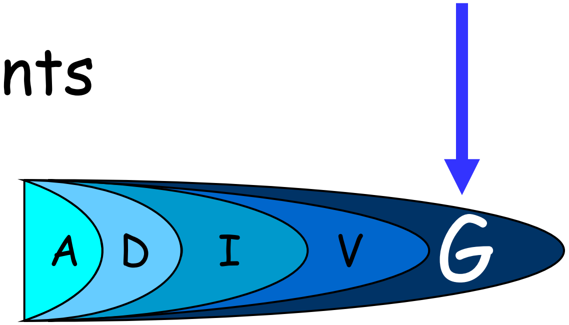
3 - Usable within a community

4 - Usable by anyone

Nature or nurture?

Two modes:

- Build and distribute libraries of reusable components
- Generalize out of program elements



(Basic distinction:

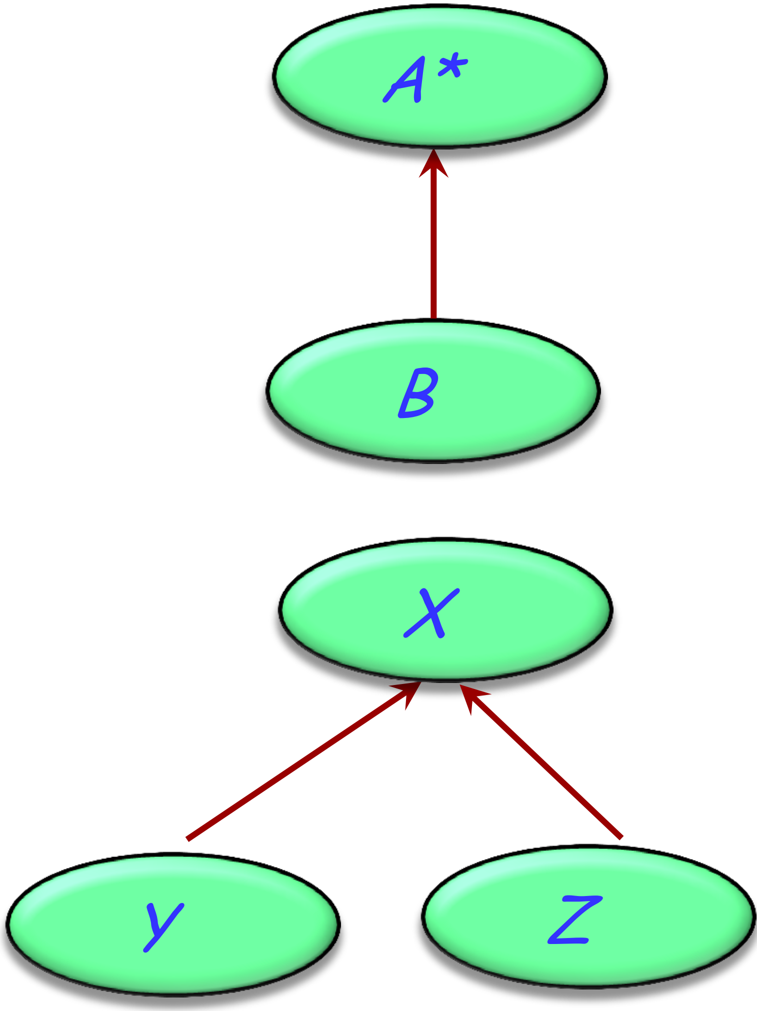
Program element --- Software component)

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

Needs management commitment



Keys to component development

Substance: Rely on a theory of the application domain

Form: Ensure consistency

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

Design principles

Object technology: Module ≡ 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)

Ordinary differential equation

```
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).

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
- “Taxomania”

Abstraction and objects

Not all classes describe "objects" in the sense of real-world things

Types of classes:

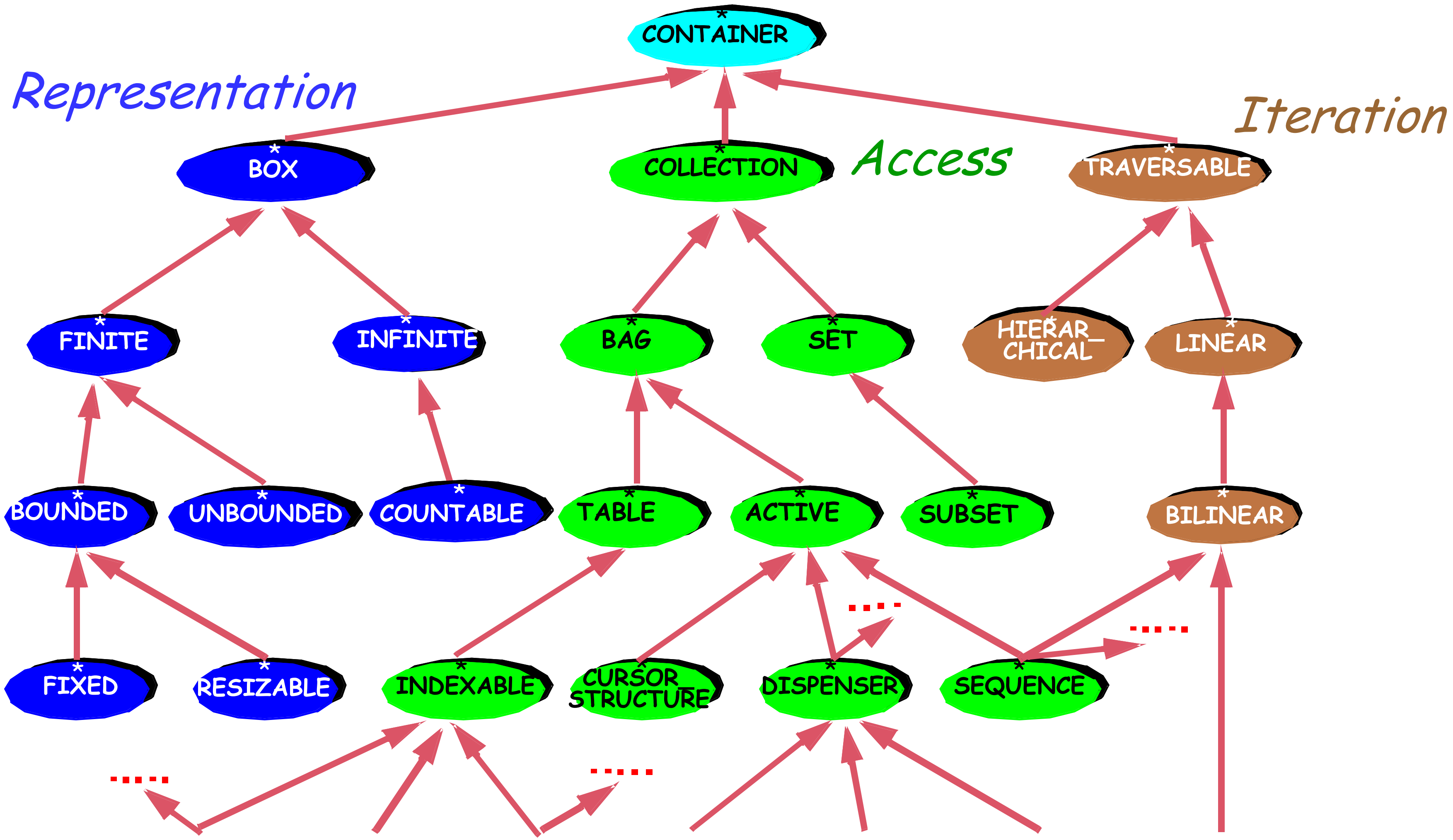
- **Analysis** classes – examples: *AIRPLANE, CUSTOMER, PARTICLE*
- **Design** classes – examples: *STATE, COMMAND, HANDLE*

  Many classes associated with design patterns fall into this category
- **Implementation** classes – examples: *ARRAY, LINKED_LIST*

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

It amounts to devising a theory of the underlying domain

Eiffelbase hierarchy



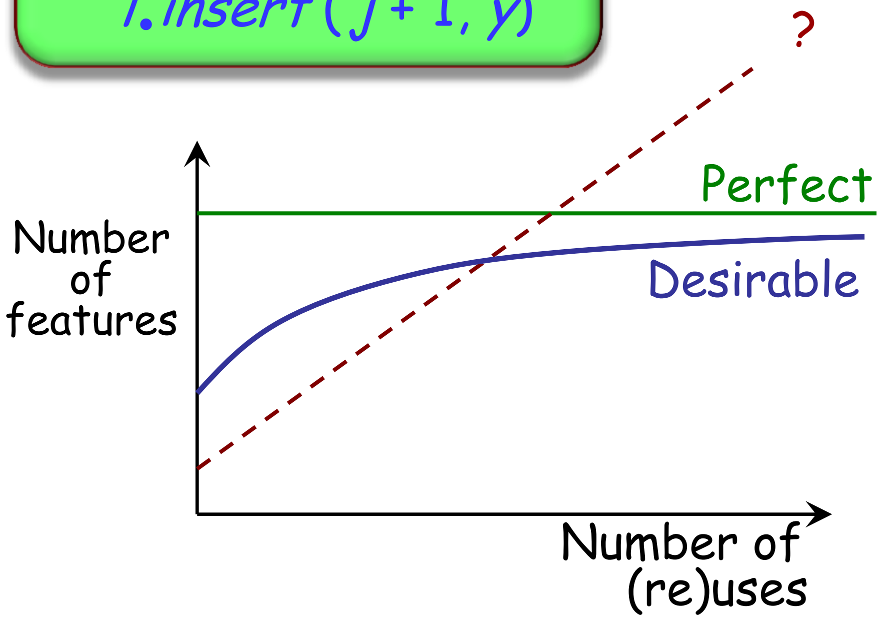
Active data structures

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 (…)

-- Typical use:
j := l.search (x)
l.insert (j + 1, y)



New interface:

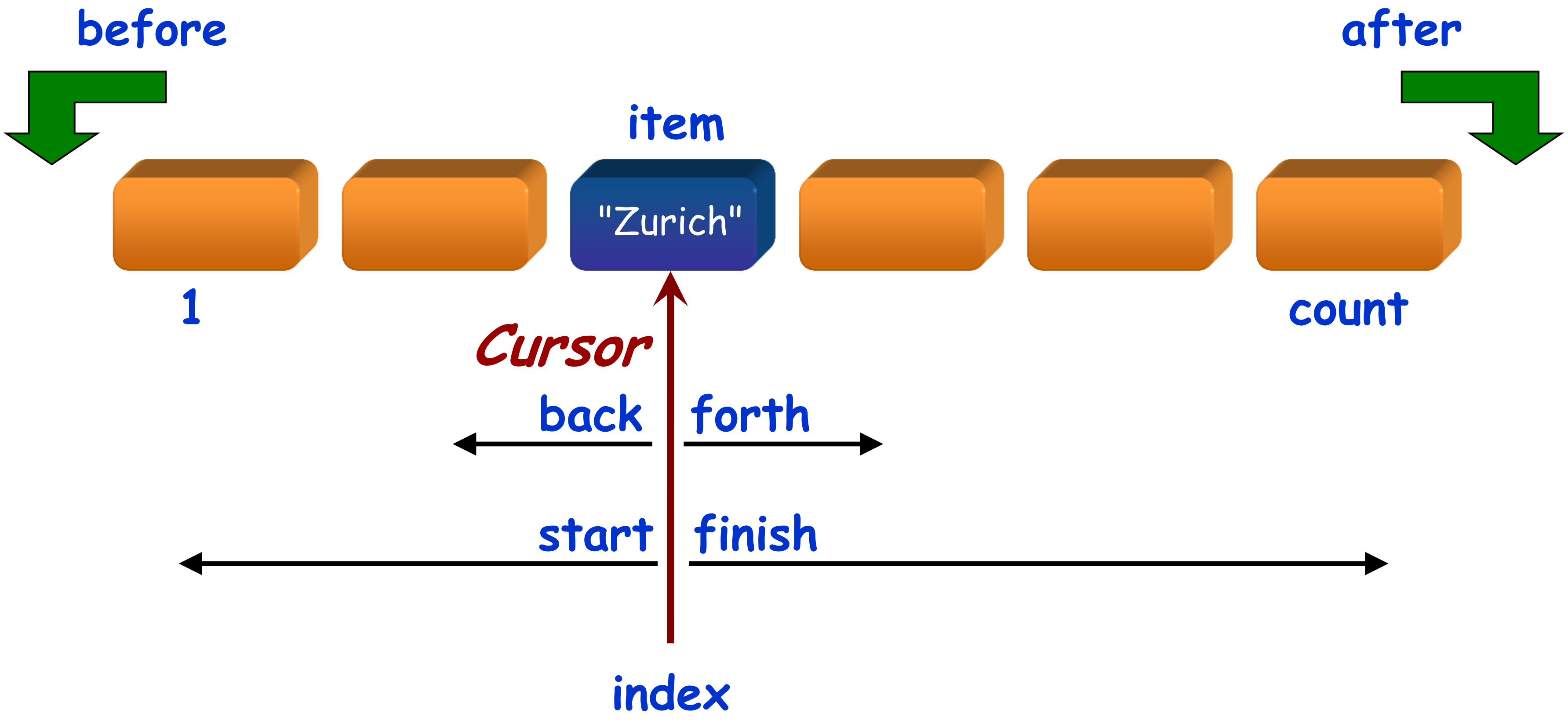
Queries:

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*

A list seen as an active data structure



Beyond internal cursors

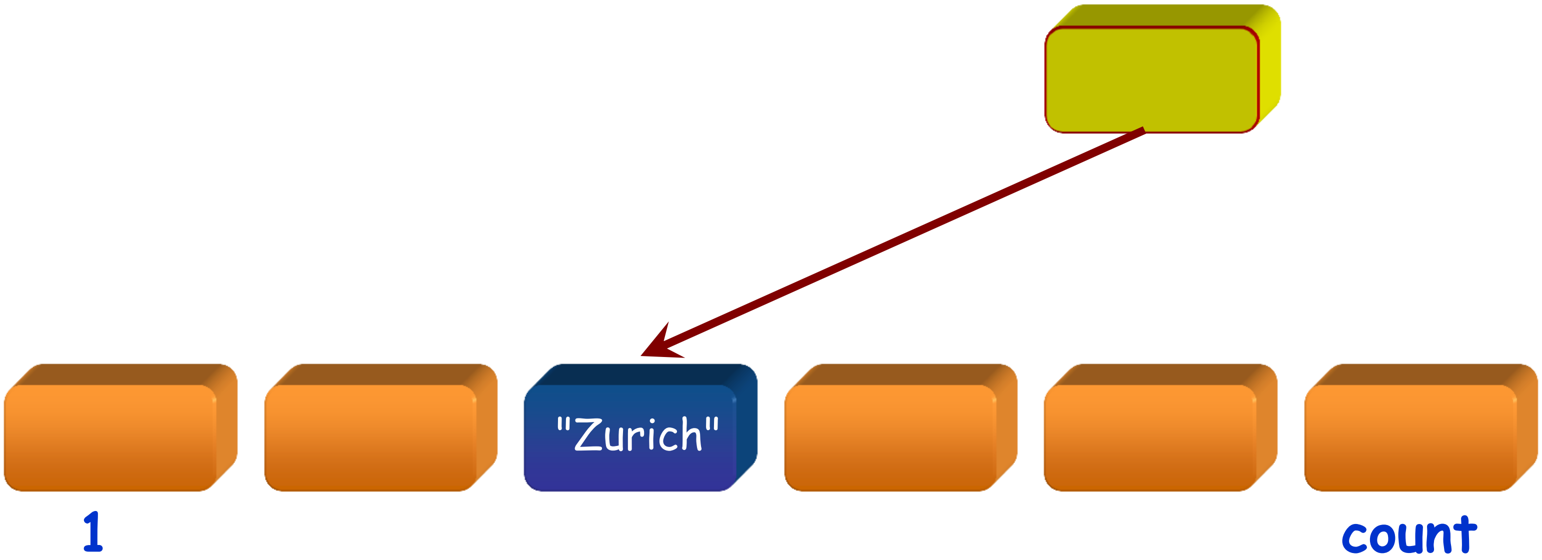
Internal cursors, as in the preceding example, have disadvantages:

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

```
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* and other commands

c.index *c.item* *c.after* 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:

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

Structure's class must be a descendant of *ITERABLE*.
This is the case 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
```

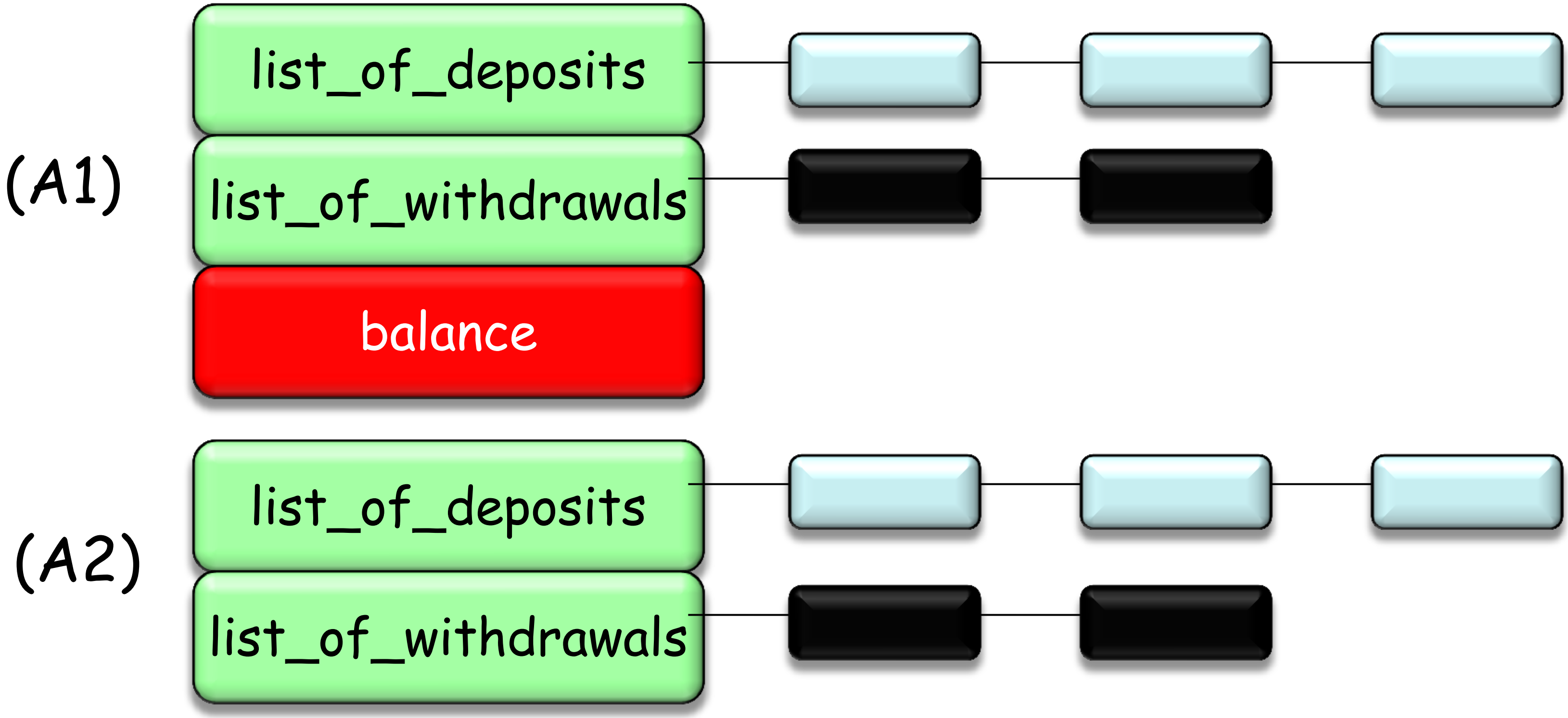
Uniform access

Uniform Access principle

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

Uniform access

balance = list_of_deposits.total - list_of_withdrawals.total



(A1)

list_of_deposits

list_of_withdrawals

balance

(A2)

list_of_deposits

list_of_withdrawals

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 * cos (theta)
                                internal_y := ro * sin (theta)
                                cartesian_available  := True
                        end
                ensure
                        cart_ok: cartesian_available
                        polar_ok: polar_available
                end
```

Representation invariant

```
invariant
        cartesian_available or polar_available
```

Accessing the horizontal coordinate

```
feature
        x: REAL
                -- Abscissa of current point
        do
                update_cartesian
                Result := x_internal
        ensure
                cartesian_ok: cartesian_available
        end
```

Adding two complex numbers

```
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

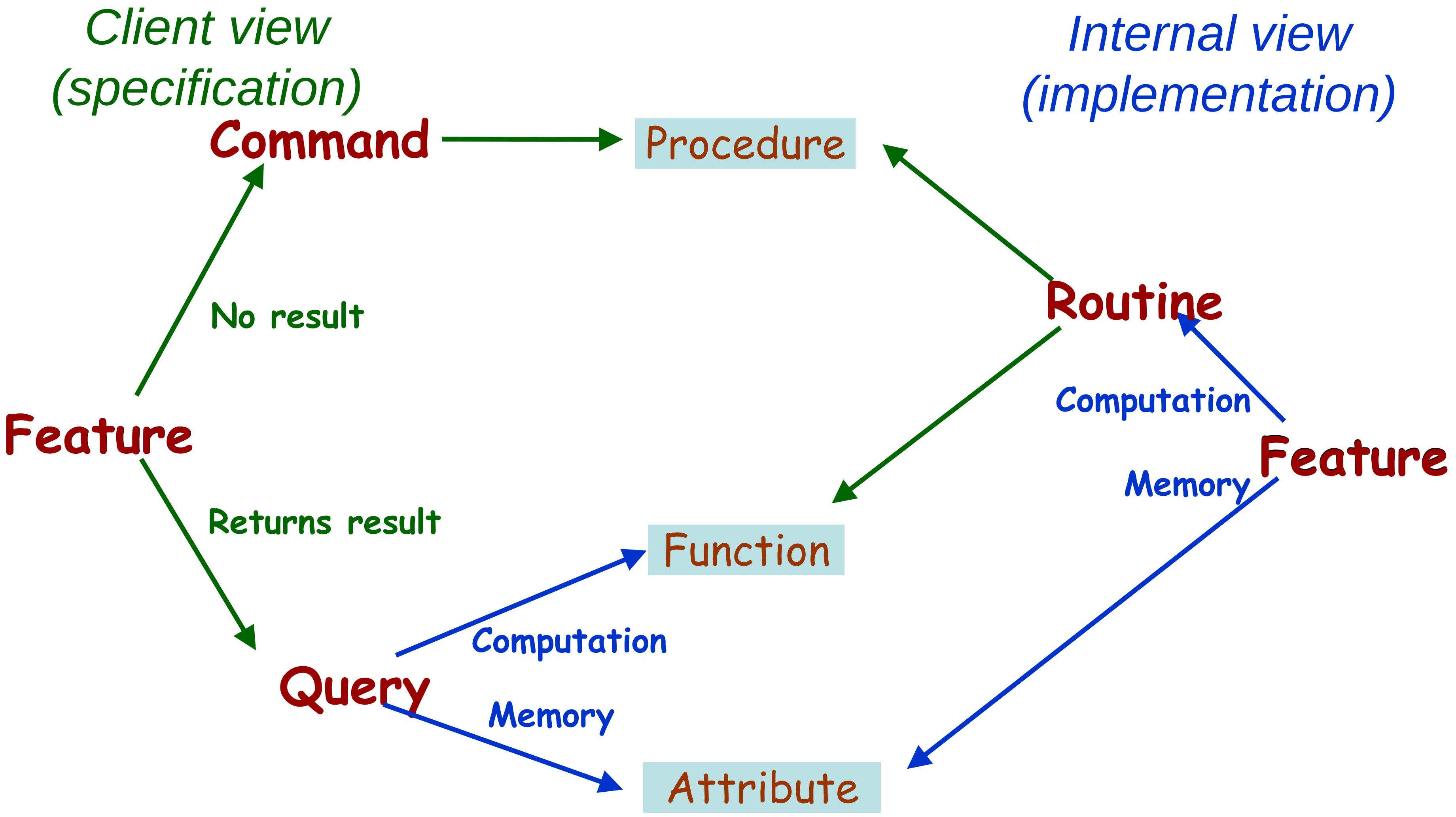
Command-Query separation principle

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)

Internal view (implementation)

Command-Query Separation Principle

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 $getint() = 2$, $getint() + getint()$ is usually not equal to 4

Command-query separation

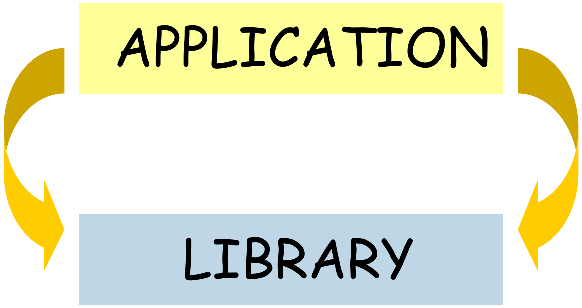
Input mechanism using EiffelBase
(instead of *n* := *getint* ()):

```
io.read_integer

n := io.last_integer
```

Libraries and contracts

Include appropriate contracts:

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



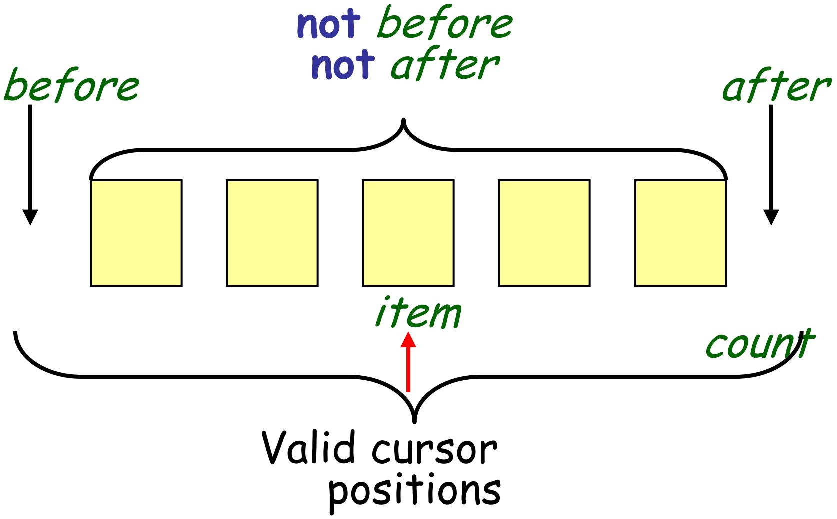
```
l.insert (x, j + k + 1)

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:

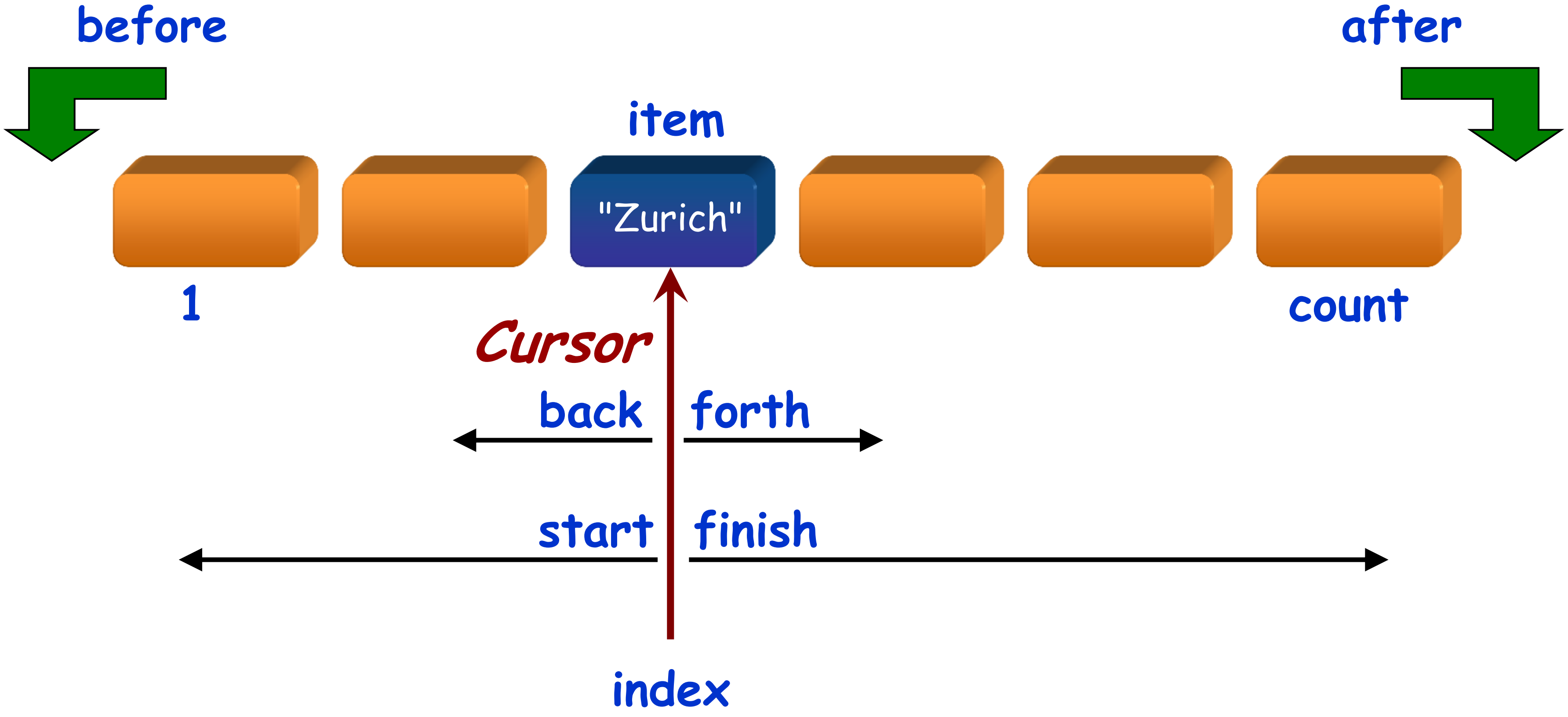
start	*finish*
forth	*back*
after	*before*



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

$$A \begin{cases} \mathit{after} = (\mathit{index} = \mathit{count} + 1) \\ \mathit{before} = (\mathit{index} = 0) \end{cases}$$

List with cursor



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:

B $\begin{cases} \textit{after} = (\textit{is\_empty} \textbf{ or } (\textit{index} = \textit{count} + 1)) \\ \textit{before} = (\textit{is\_empty} \textbf{ or } (\textit{index} = 0)) \end{cases}$

-- Hence: *is_empty* = (*before* **and** *after*)

Symmetric but unpleasant. Leads to frequent tests

if *after* **and not** *is_empty* **then** ...

instead of just

if *after* **then** ...

Introducing sentinel items

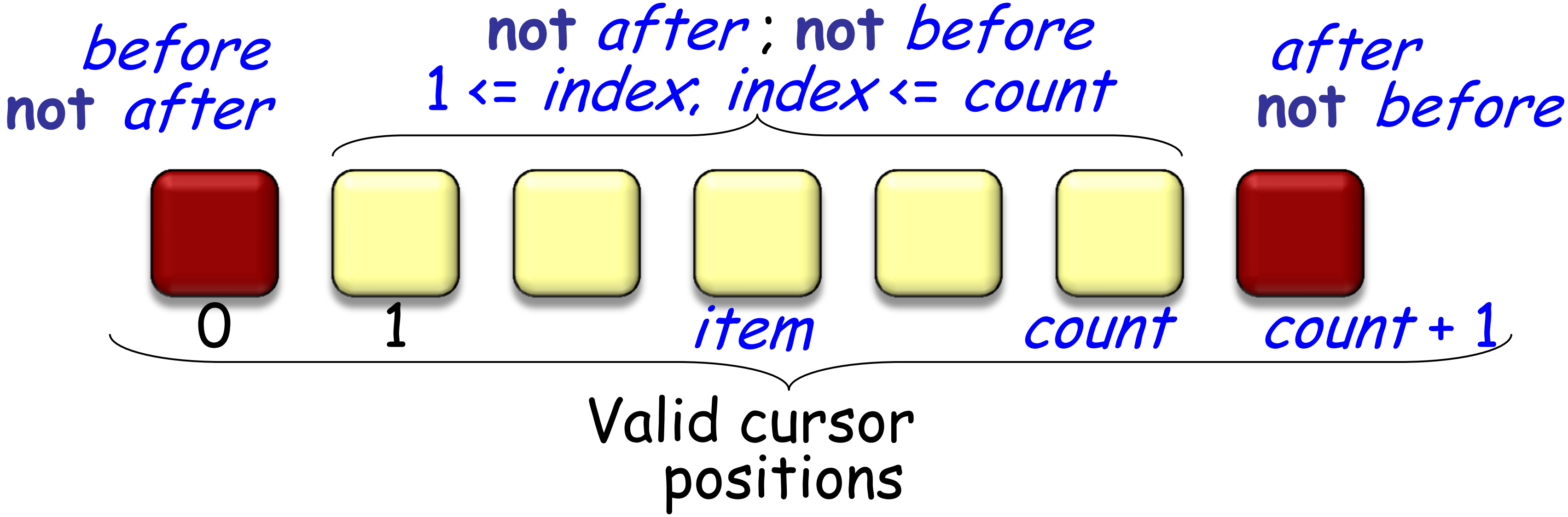
Invariant (partial):

0 <= *index*

index <= *count* + 1

$$A \left\{ \begin{array}{l} before = (index = 0) \\ after = (index = count + 1) \end{array} \right.$$

not (*after* **and** *before*)



before
not *after*

not *after* ; **not** *before*
1 <= *index*; *index* <= *count*

after
not *before*

0

1

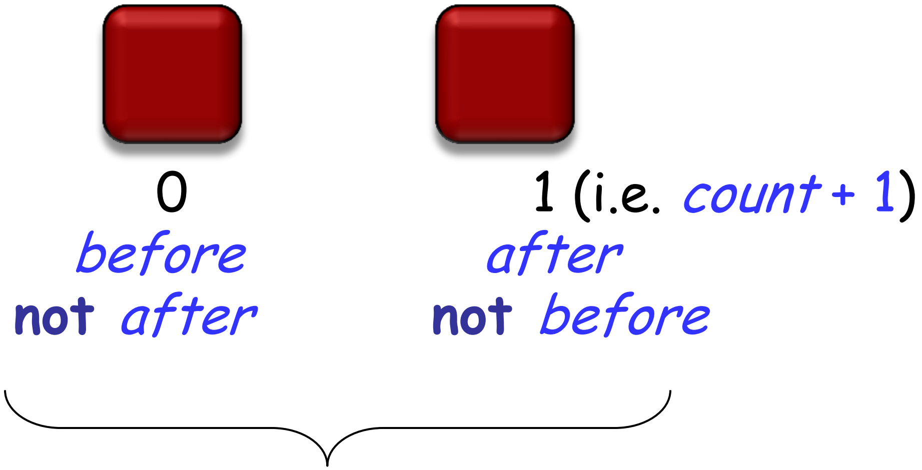
item

count

count + 1

Valid cursor positions

The case of an empty structure



0

before

not *after*

1 (i.e. *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):

```
put
    require
        not full
    do
        ...
    ensure
        ...
    end
```

How big should a class be?

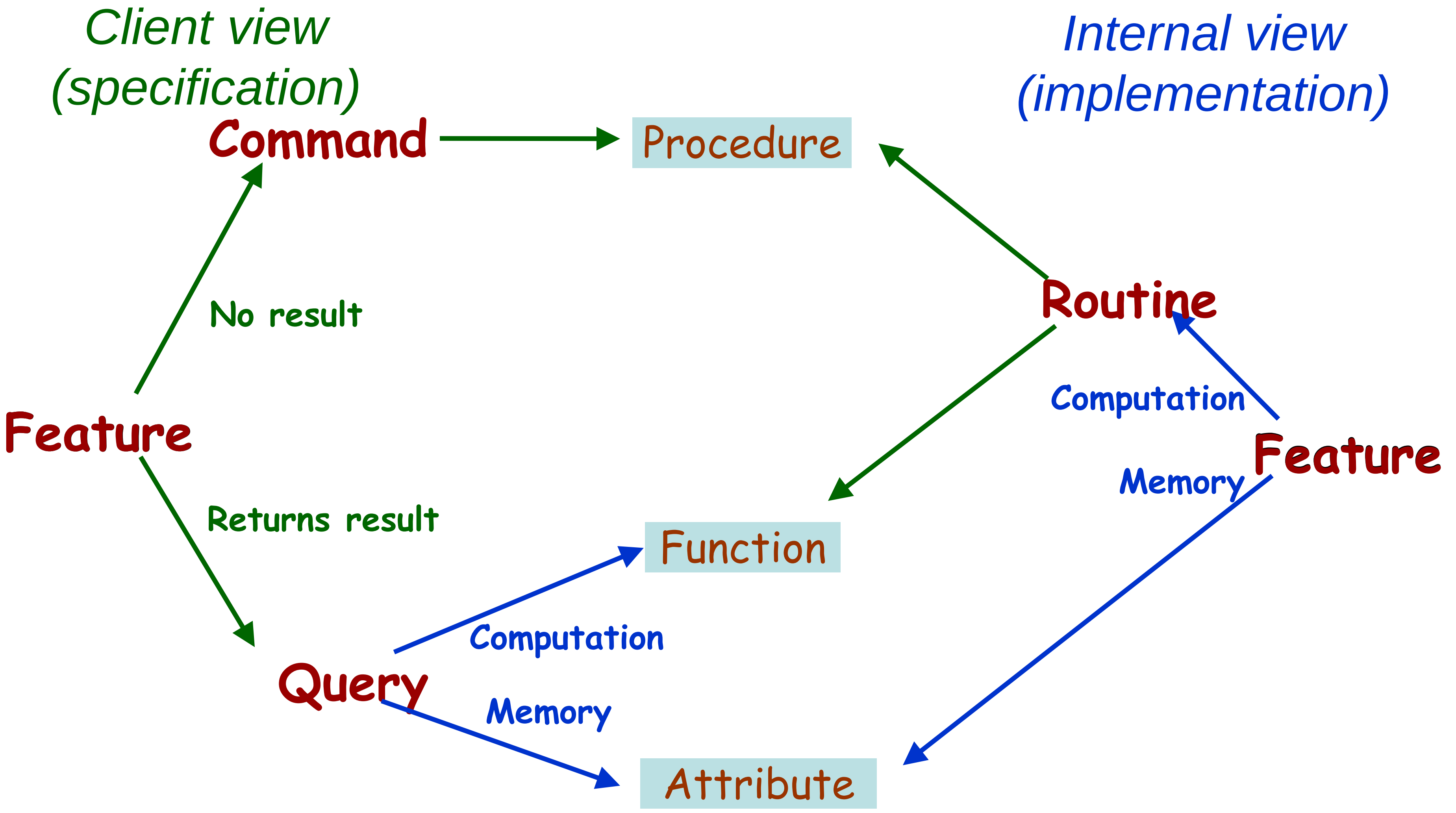
The first question is how to measure class size. Candidate metrics:

- 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.

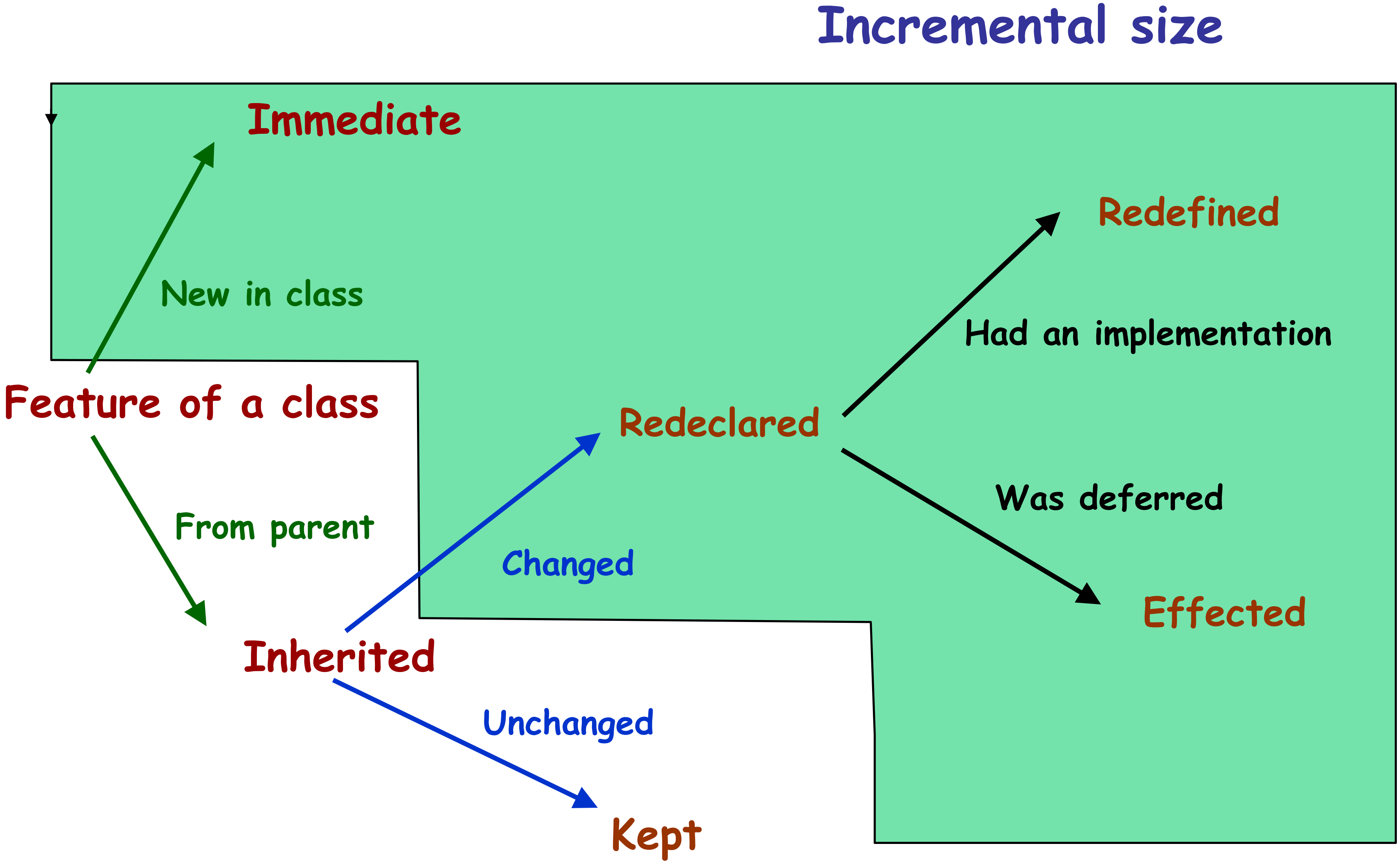
Feature classification (reminder)



Client view (specification)

Internal view (implementation)

Another classification



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

0 to 5 features	43
6 to 10 features	14
11 to 15 features	10
16 to 20 features	4
21 to 40 features	17
41 to 80 features	9
81 to 142 features	2

(All measures from version 6.0, courtesy Yi Wei)

EiffelVision on Windows

Percentages, rounded.
733 classes, 5872 exported features

0 to 5 features	64
6 to 10 features	14
11 to 15 features	8
16 to 20 features	5
21 to 40 features	7
41 to 80 features	2

EiffelVision on Linux

Percentages, rounded.
698 classes, 8614 exported features

0 to 5 features	63
6 to 10 features	13
11 to 15 features	8
16 to 20 features	5
21 to 40 features	8
41 to 80 features	2

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:

Number of features	4408
Percentage of queries	66%
Percentage of commands	34%
Average number of arguments to a feature	0.5
Maximum number	5
No arguments	57%
One argument	36%
Two arguments	6%
Three or more arguments	1%

Size of feature interfaces

Including non-exported features:

Average number of arguments to a feature	0.6
Maximum number	12
No arguments	55%
One argument	36%
Two arguments	7%
Three arguments	2%
Four arguments	0.4%
Five or six arguments	0.1%

Size of feature interfaces

EiffelVision on Windows (733 classes, exported only)

Number of features	5872
Percentage of queries	56%
Percentage of commands	44%
Average number of arguments to a feature	0.5
Maximum number	10
No argument	67%
One argument	23%
Two arguments	6%
Three arguments	1.5%
Four arguments	1.5%
Five to seven arguments	0.6%

Size of feature interfaces

EiffelVision on Linux (698 classes, exported only)

Number of features	8614
Percentage of queries	56%
Percentage of commands	44%
Average number of arguments to a feature	0.96
Maximum number	14
No argument	49%
One argument	28%
Two arguments	15%
Three arguments	4%
Four arguments	2%
Five to seven arguments	1%

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:

```
my_window.set_background_color ("blue")
...
my_window.display
```

Operands and options

Useful checklist for options:

Option	Default	Set	Accessed
Window color	White	*set_background_color*	*background_color*
Hidden?	No	*set_visible* *set_hidden*	*hidden*

Naming (classes, features, variables…)

Traditional advice (for ordinary application programming):

- Choose meaningful variable names!

New and old names for EiffelBase classes

Class	Features		
ARRAY	*put*	*item* *enter*	*entry*
STACK	*put*	*item* *push*	*remove* *top*
QUEUE	*put* *add*	*item* *oldest*	*remove* *remove_oldest*
HASH_TABLE	*put*	*item* *insert* *value*	*remove* *delete*

Naming rules

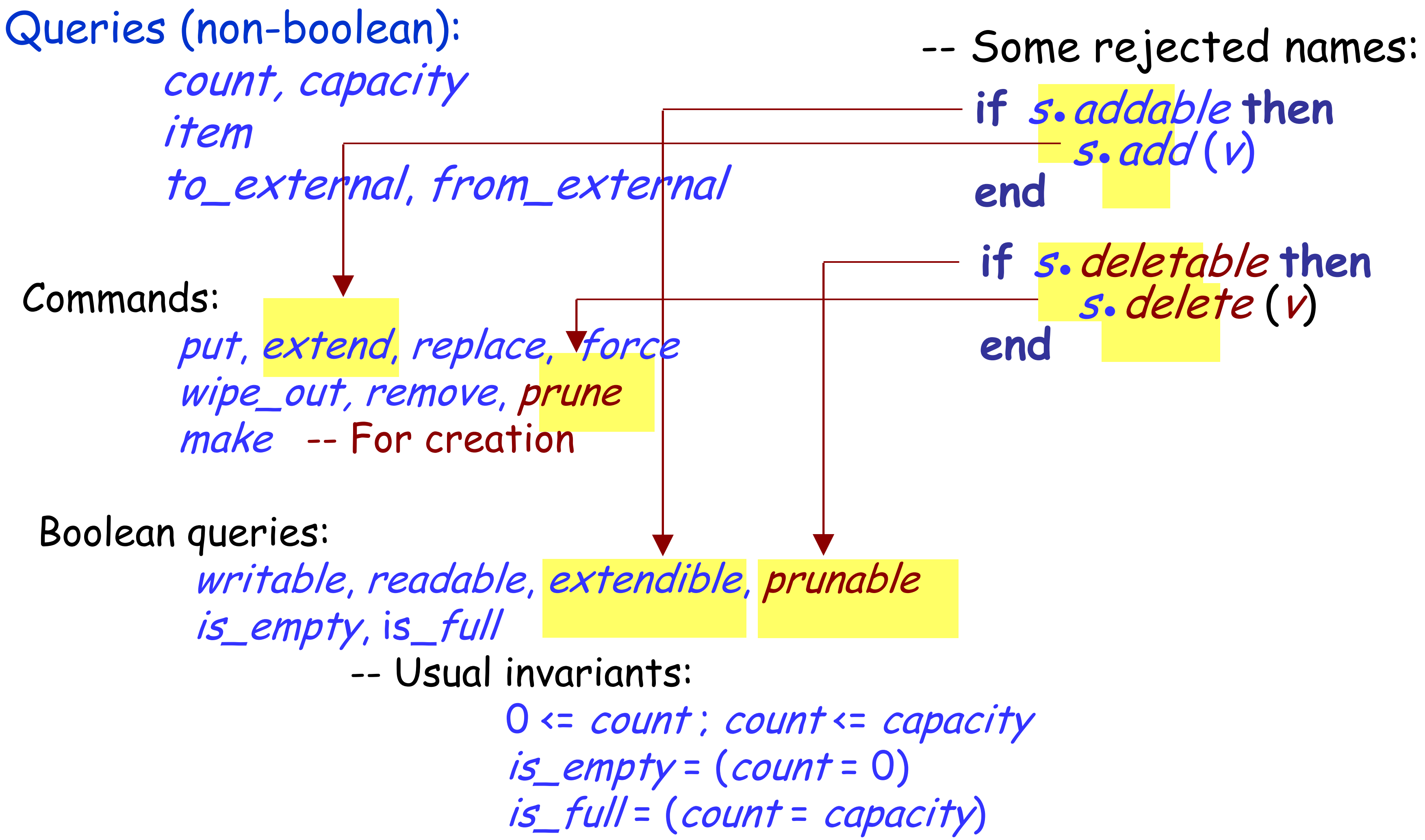
Achieve consistency by systematically using a set of standardized names.

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

Commands:

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

Boolean queries:

writable, readable, extendible, prunable
is_empty, is_full

-- Usual invariants:

0 <= count ; count <= 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

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

Obsolete features and classes

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*:

```
enter (i: V; v: T)
        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
```

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