Must it be assumed that because we are engineers beauty is not our concern, and that while we make our constructions robust and durable we do not also strive to make them elegant?

Is it not true that the genuine conditions of strength always comply with the secret conditions of harmony?

The first principle of architectural esthetics is that the essential lines of a monument must be determined by a perfect adaptation to its purpose.

Gustave Eiffel, 1887

From his response in the newspaper *Le Temps* **to a petition by members of the literary and artistic Establishment protesting his project of elevating a tower of iron in Paris.**

THE KEY IMMEDIATE BENEFITS

- The libraries: building software from quality components.
- Reliability.
- Ease of change.
- Sharing, communication, standardization between developers.
- Ease of switching over; compatibility with existing software.
- Reuse.

THE EIFFEL METHOD

Structure

Modularity

Use of libraries

Safety through typing

Design by contract

Systematic use of inheritance

Attention to detail: style, names, comments, documentation

THE B.O.N. METHOD (BUSINESS OBJECT NOTATION)
Jean-Marc Nerson (SOL, Paris), Kim Waldén (Enea, Stockholm)

APPLYING DESIGN BY CONTRACT TO ANALYSIS .

BON provides a clear notation and methodological guidelines for high-level analysis and design, in the Eiffel spirit of precision, scalability and clarity. Three key concepts: seamlessness, reversibility and software contracting.

- **Well-defined set of conventions.**
- **Supports semantics (assertions, ...), not just structure.**
- **Mechanisms for systematic development; supports Design by Contract.**
- **Textual as well as graphical variants. Three views: graphics (bubbles and arrows!), tables, formal text (Eiffel-like).**
- **Meant for use with software tools (EiffelCase)**
- **SCALES UP: Abstraction and grouping facilities: Classes, Clusters, entire systems. Zoom in, zoom out, abstract**
- **Supporting tool: EiffelCase.**

EiffelCase

The full lifecycle tool for software modeling, architectural design, and user-developer communication.

Supports a seamless approach to software development.

Multiple views:

- **Type charts.**
- **Graphics.**
- **Formal text (Eiffel).**
- **Informal text.**

Automatic update, guaranteeing compatibility between views.

EiffelStore

Uniform persistence mechanism.

- **Basic Eiffel facilities:** *STORABLE* **and** *ENVIRONMENT***.**
- **Relational database interface: object ↔ record and class ↔ relation. Currently: Oracle and Sybase. Source compatibility via the "handle" mechanism.**
- **Object-oriented database interfaces under development.**

MORE ABSTRACTIONS

STATE

- **Catalog, editor**
- $CONTEXT \times EVENT \underset{f}{\rightarrow} RESPONSE$
- **Curried:** $CONTEXT \underset{f}{\rightarrow} (EVENT \underset{f}{\rightarrow} RESPONSE)$
- **i.e.** $CONTEXT \underset{f}{\rightarrow} BEHAVIOR$

RESPONSE

- $COMMAND \times TRANSITION\_LABEL$

COMMAND

- **Catalog, editor**
- **Examples: *Save, Open*.**
- **Undoable or not.**
- **Automatic history and undo-redo mechanism.**

DIALOG ABSTRACTIONS

APPLICATION

- **Catalog, editor**
- **State transition diagram**
- **Initial state, final state**
- **Sub-application**

EVENT

- **Catalog, editor**
- **Example: *ACTIVATE BUTTON***

CONTEXT

- **Catalog, editor**
- **Examples: *Push_button, Window.***

EiffelBuild

More than a pretty interface.™

Convenient, simple tool for building entire interactive applications.

Key abstractions:

- **Event**
- **Context**
- **Command (undoable or not)**
- **Response**
- **Behavior**
- **Transition**
- **State**
- **Application**

EiffelVision

The portable library for graphical applications and advanced user interfaces.

Full source compatibility between toolkits: Motif, Open Look, Windows (later MacOS, Presentation Manager).

Graphics (world/figures/windows) and user interface objects, all in the same framework.

Simple set of clear abstractions. No need to learn details of X, Motif, UIL, Windows etc. Single-language framework.

Toolkit adaptability: currently Motif and Open Look.

Changing toolkit:

- **Statically: no source code change.**
- **Dynamically: change handle**

  !OPEN_LOOK! handle

  !MOTIF! handle

  !WINDOWS! handle

LIBRARIES

- **EiffelBase (fundamental data structures and algorithms, iteration etc.).**
- **EiffelVision (graphics, GUI).**
- **EiffelLex (lexical analysis).**
- **EiffelParse (parsing).**
- **EiffelStore (database access).**
- **EiffelNet (client-server communication).**

Linnaean reconstruction of software fundamentals

Reusable Software:
The Base Object-Oriented Component Libraries
Prentice Hall, 1994

About 1500 classes, 15,000 visible features

Full use of the O-O mechanisms: assertions, multiple and repeated inheritance, constrained and unconstrained ssgenericity etc.

Finalization and performance of generated code

Final compilation generates a C package, complete with make file and performs optimizations.

Optimizations in final mode:

- **Unneeded routine removal.**
- **Call simplification (static binding) if legitimate.**
- **in-line routine expansion.**
- **Array optimization.**

Array performance on array-intensive computations in version 3.3 is within 0% to 7% of C (depending on the platform).

FAST RECOMPILATION IN EIFFELBENCH

Aim: make sure that a "system" is up-to-date.

- Recompile only as needed. Automatic dependency and change analysis based on client and inheritance relations.
- No make files. Process is entirely automatic.
- Full type checking in all cases.

Three forms of compilation:

- Melting.
- Freezing.
- Final-compile.

Melting is the fastest. Finalizing generates the fastest code. Freezing is required whenever external (e.g. C) software is added or changed.

Final compilation generates a C package, complete with make file and performs optimizations.

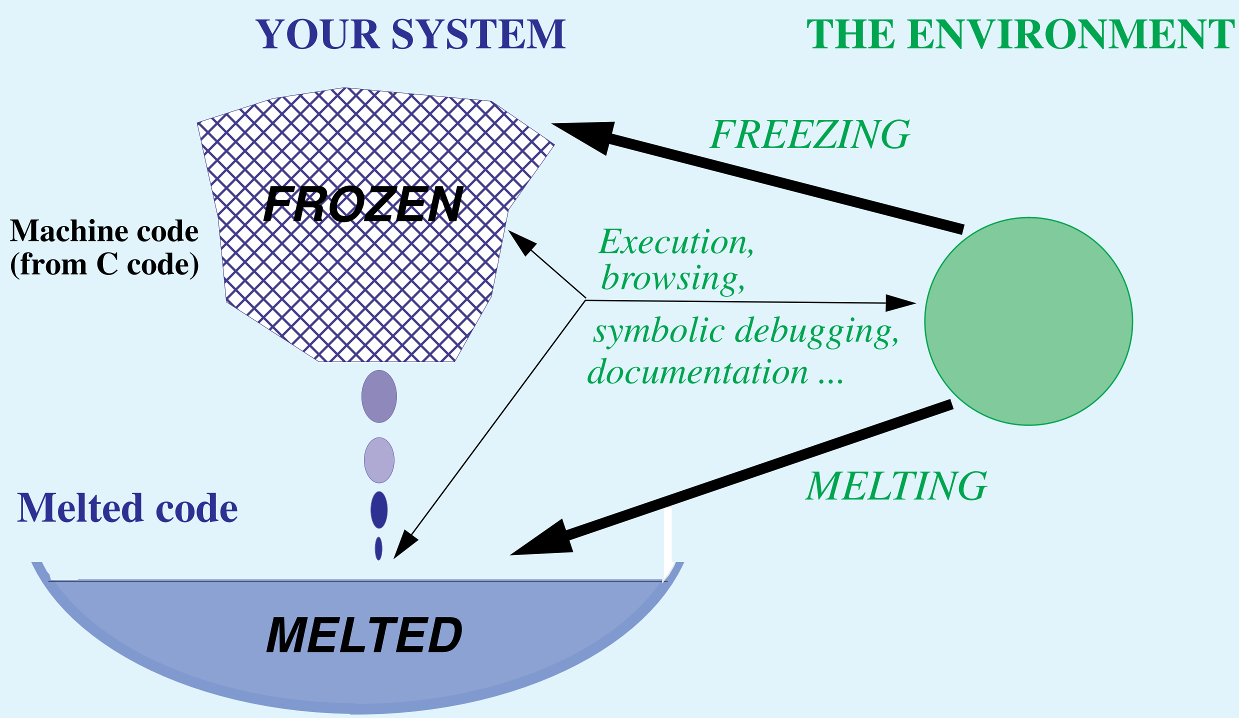
The importance of speedy recompilation

	Small system	Large system
Small change		The key case
Big change		

INCREMENTAL DEVELOPMENT: THE MELTING ICE TECHNOLOGY

Combination of interpreted and compiled code. Get the best of both worlds.

Changes "melt" compiled code; the changed parts will be interpreted rather than compiled.



ENVIRONMENT PRINCIPLES

5. Typed Environment

In an object-oriented environment supporting a statically typed language and method, the environment's visual conventions should display and enforce the type constraints on development objects.

The TYPED DRAG AND DROP model:

- **Pebbles (stones).**
- **Holes.**

ENVIRONMENT PRINCIPLES

4. Semantic Consistency

An object-oriented environment must enable its users, for any symbol (textual, graphical or otherwise) representing a development object in the user interface, to select the object through its symbol and apply any operation that is semantically valid for the object, regardless of the symbol's context — tool, location, format, representation.

The visual representation fully represents the object: the object's semantics is available through the representation — text, icon, pebble.

ENVIRONMENT PRINCIPLES

3. Object-Oriented Tools

In an object-oriented environment, each tool must be based on a developer's object type, not on a type of operation.

- **Class Tool.**
- **Feature Tool.**
- **System Tool.**
- **Project Tool.**
- **Explanation Tool.**

ENVIRONMENT PRINCIPLES

2. Data Abstraction:

In an object-oriented environment, the basic way of working must be through direct manipulation of visual representations of developer's object types.

DEVELOPER'S OBJECT TYPES:

- **Class.**
- **Feature, Routine.**
- **System.**
- **System.**
- **Project.**
- **System.**
-

ENVIRONMENT PRINCIPLES

1. Method-Environment Consistency

A development environment meant to support a particular method or language must rely on a consistent set of user interaction conventions which closely parallel the concepts promoted by the method or language.

Examples:

- **Lisp environments.**
- **C environments.**
- **Object-oriented environments.**

AN OBJECT-ORIENTED ENVIRONMENT MUST BE BASED ON DATA ABSTRACTIONS

OBJECT TOOLS:

- **Class.**
- **Feature (routine, method).**
- **System (assembly of class – closest equivalent to "program").**
- **Project.**
- **Explanation.**

All objects, and hence all tools, are typed.

Every typed object/tool has a pebble ready to be dropped into matching holes.

TRADITIONAL ENVIRONMENTS (Including for O-O languages & methods)have FUNCTION TOOLS:

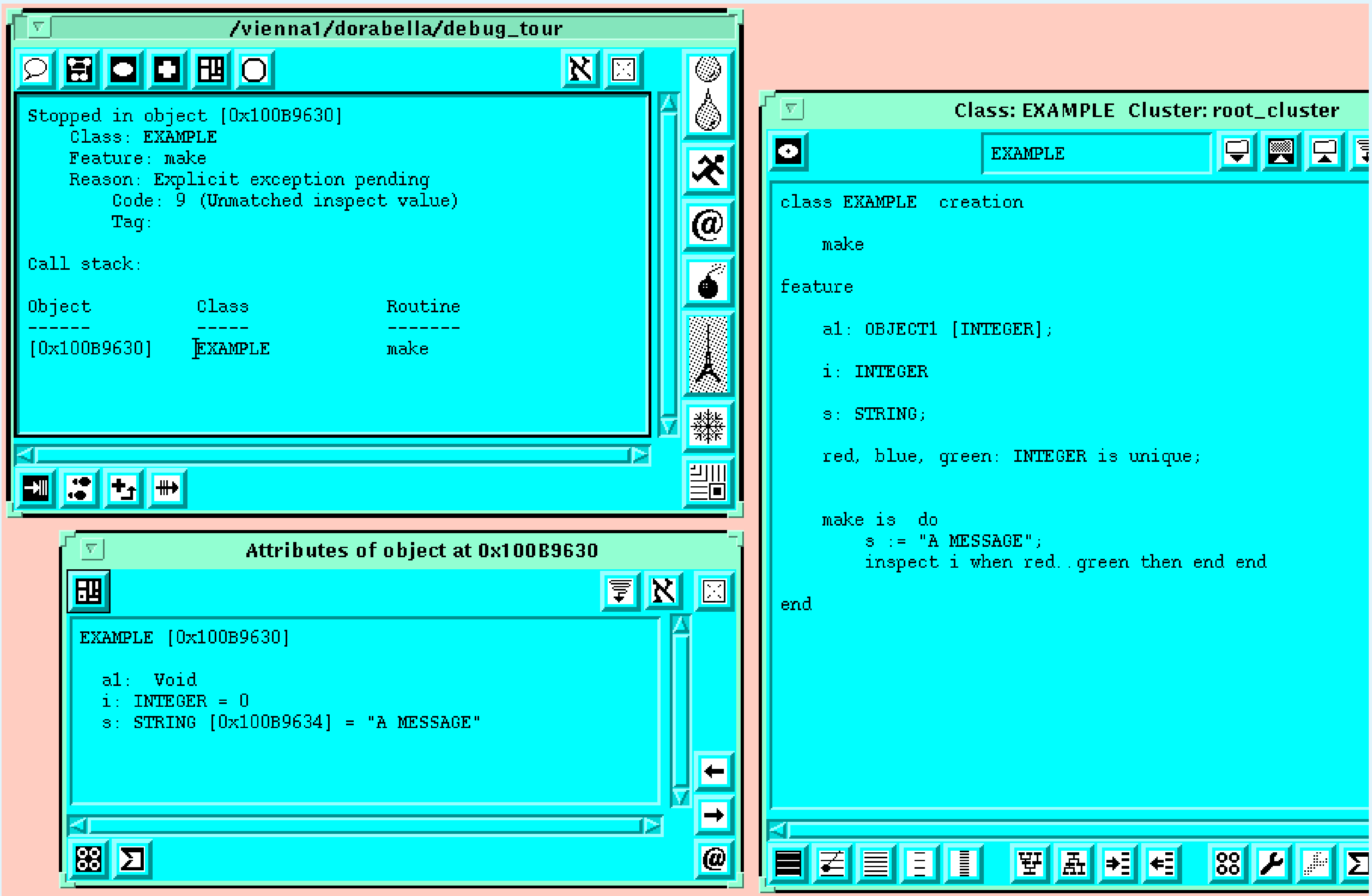
- Compiler
- Interpreter
- Debugger
- Browser
- Tester
- Prototyper
- Interface emulator
- ...

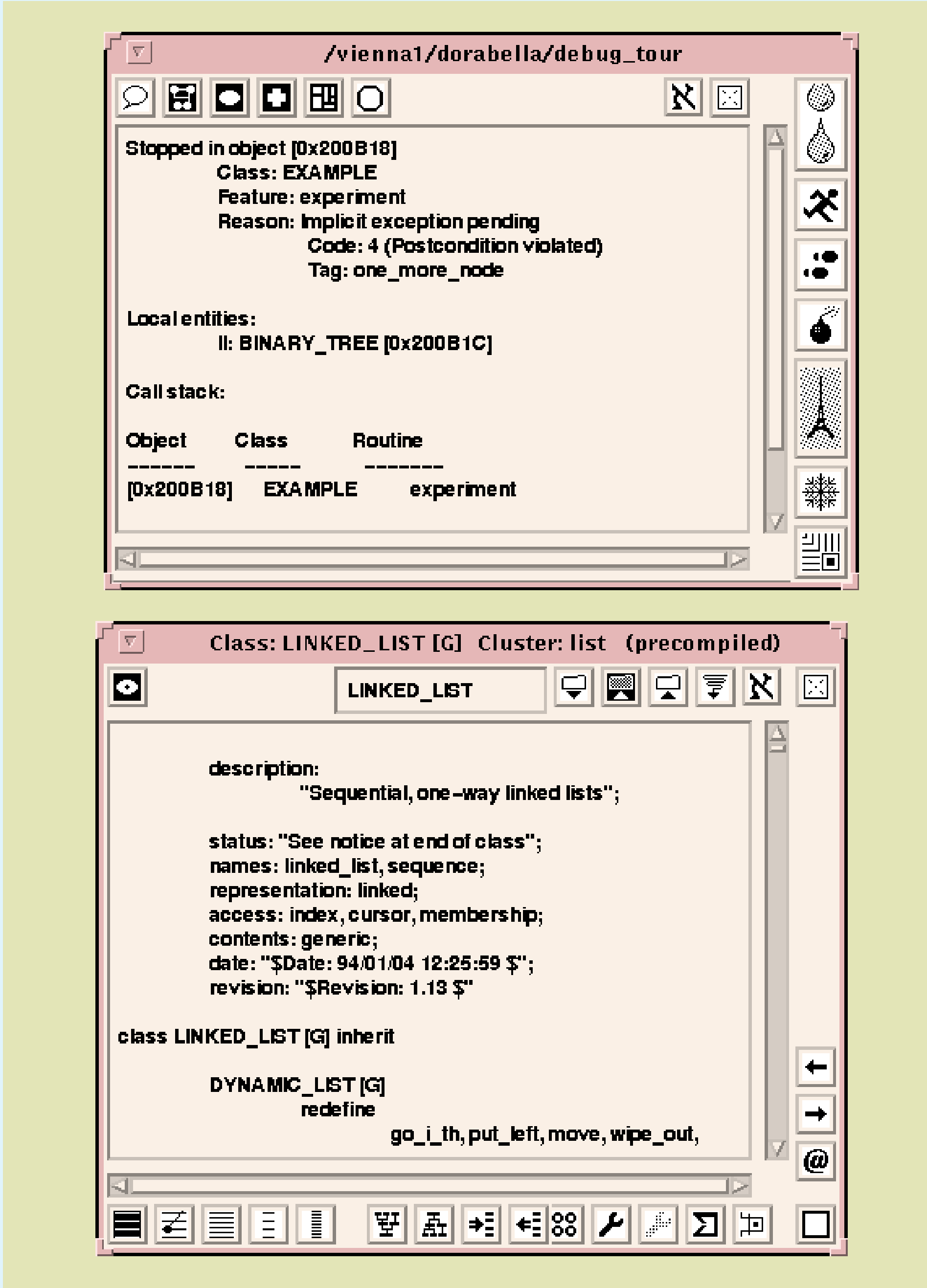
WHAT DOES IT MEAN TO HAVE AN "OBJECT-ORIENTED ENVIRONMENT"?

Vulgar meaning: anything that has a picture in it.

In reality:

An object-oriented environment is a development environment which supports the construction of software based on the direct visual manipulation of formally defined abstract notions.


/vienna1/dorabella/debug_tour
Stopped in object [0x100B9630]
Class: EXAMPLE
Feature: make
Reason: Explicit exception pending
Code: 9 (Unmatched inspect value)
Tag:
Call stack:
Object Class Routine
[0x100B9630] EXAMPLE make
Attributes of object at 0x100B9630
EXAMPLE [0x100B9630]
a1: Void
i: INTEGER = 0
s: STRING [0x100B9634] = "A MESSAGE"
Class: EXAMPLE Cluster: root_cluster
EXAMPLE
class EXAMPLE creation
make
feature
a1: OBJECT1 [INTEGER];
i: INTEGER
s: STRING;
red, blue, green: INTEGER is unique;
make is do
s := "A MESSAGE";
inspect i when red..green then end end
end


/vienna1/dorabella/debug_tour
Stopped in object [0x200B18]
Class: EXAMPLE
Feature: experiment
Reason: Implicit exception pending
Code: 4 (Postcondition violated)
Tag: one_more_node
Local entities:
ll: BINARY_TREE [0x200B1C]
Call stack:
Object Class Routine
[0x200B18] EXAMPLE experiment
Class: LINKED_LIST [G] Cluster: list (precompiled)
LINKED_LIST
description:
"Sequential, one-way linked lists";
status: "See notice at end of class";
names: linked_list, sequence;
representation: linked;
access: index, cursor, membership;
contents: generic;
date: "$Date: 94/01/04 12:25:59 $";
revision: "$Revision: 1.13 $"
class LINKED_LIST [G] inherit
DYNAMIC_LIST [G]
redefine
go_i_th, put_left, move, wipe_out,

EiffelBench

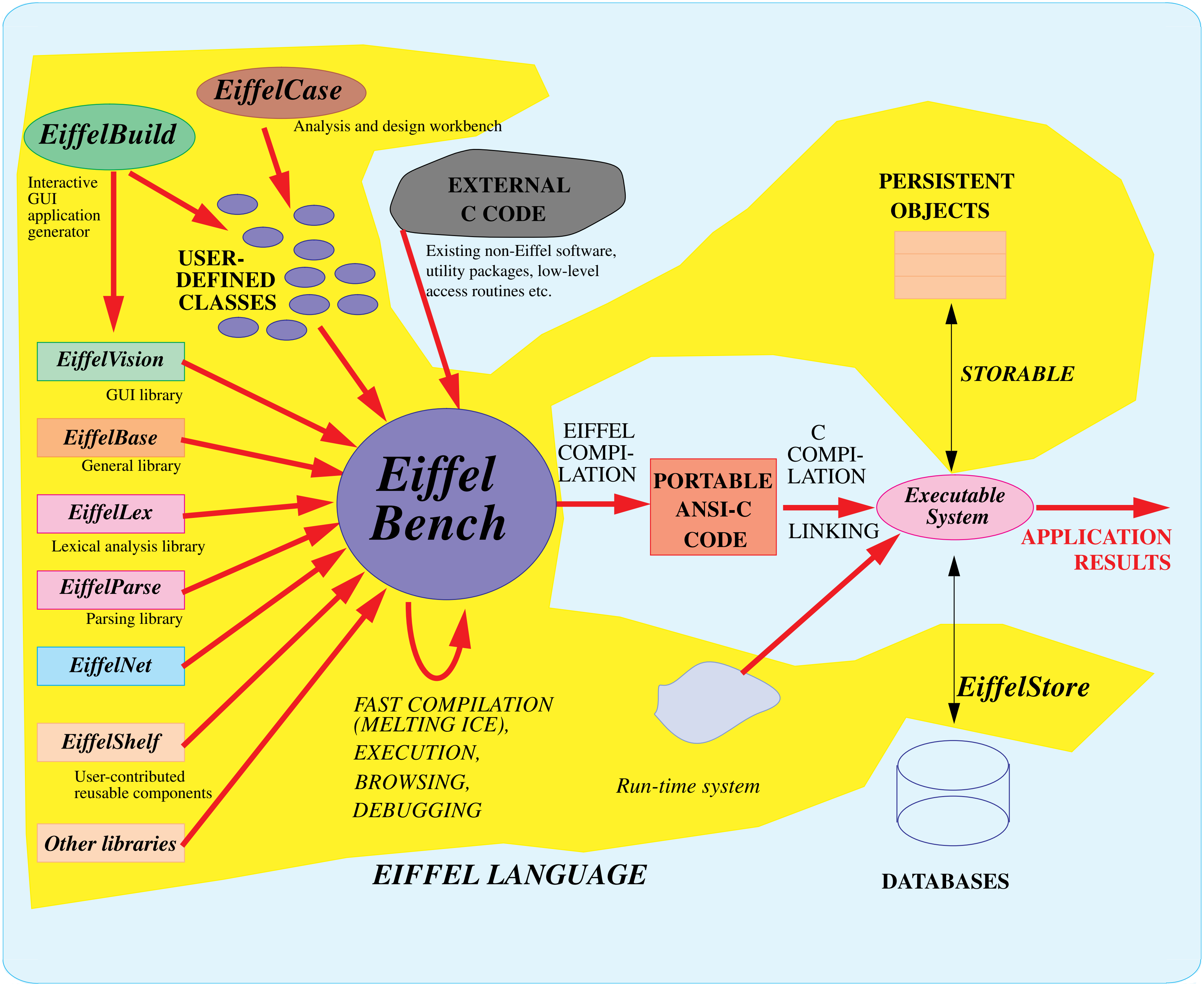
Graphical environment for developing O-O software.

Built with EiffelVision: portable across toolkits.

Truly object-oriented: system tool, class tool, routine tool, object tool...

Browsing, debugging, testing, documenting.

Basic metaphor: bring a typed pebble into a hole in which the pebble fits.


EiffelCase
Analysis and design workbench
EiffelBuild
Interactive GUI application generator
EXTERNAL C CODE
Existing non-Eiffel software, utility packages, low-level access routines etc.
PERSISTENT OBJECTS
USER-DEFINED CLASSES
EiffelVision
GUI library
EiffelBase
General library
EiffelLex
Lexical analysis library
EiffelParse
Parsing library
EiffelNet
EiffelShelf
User-contributed reusable components
Other libraries
Eiffel Bench
EIFFEL COMPI-LATION
PORTABLE ANSI-C CODE
C COMPI-LATION
LINKING
STORABLE
Executable System
APPLICATION RESULTS
FAST COMPILATION (MELTING ICE),
EXECUTION,
BROWSING,
DEBUGGING
Run-time system
EiffelStore
DATABASES
EIFFEL LANGUAGE

ISE EIFFEL 3: THE COMPONENTS

EIFFELBENCH

Graphical development environment. Compiler/interpreter, browsing, execution, debugging...

EIFFELBASE

Key libraries.

EIFFELVISION

Graphics and user interfaces.

EIFFELLEX, EIFFELPARSE

Language analysis.

EIFFELSTORE

Standardized database interface.

EIFFELBUILD

Interactive application builder.

EIFFELNET

Client-server communication library.

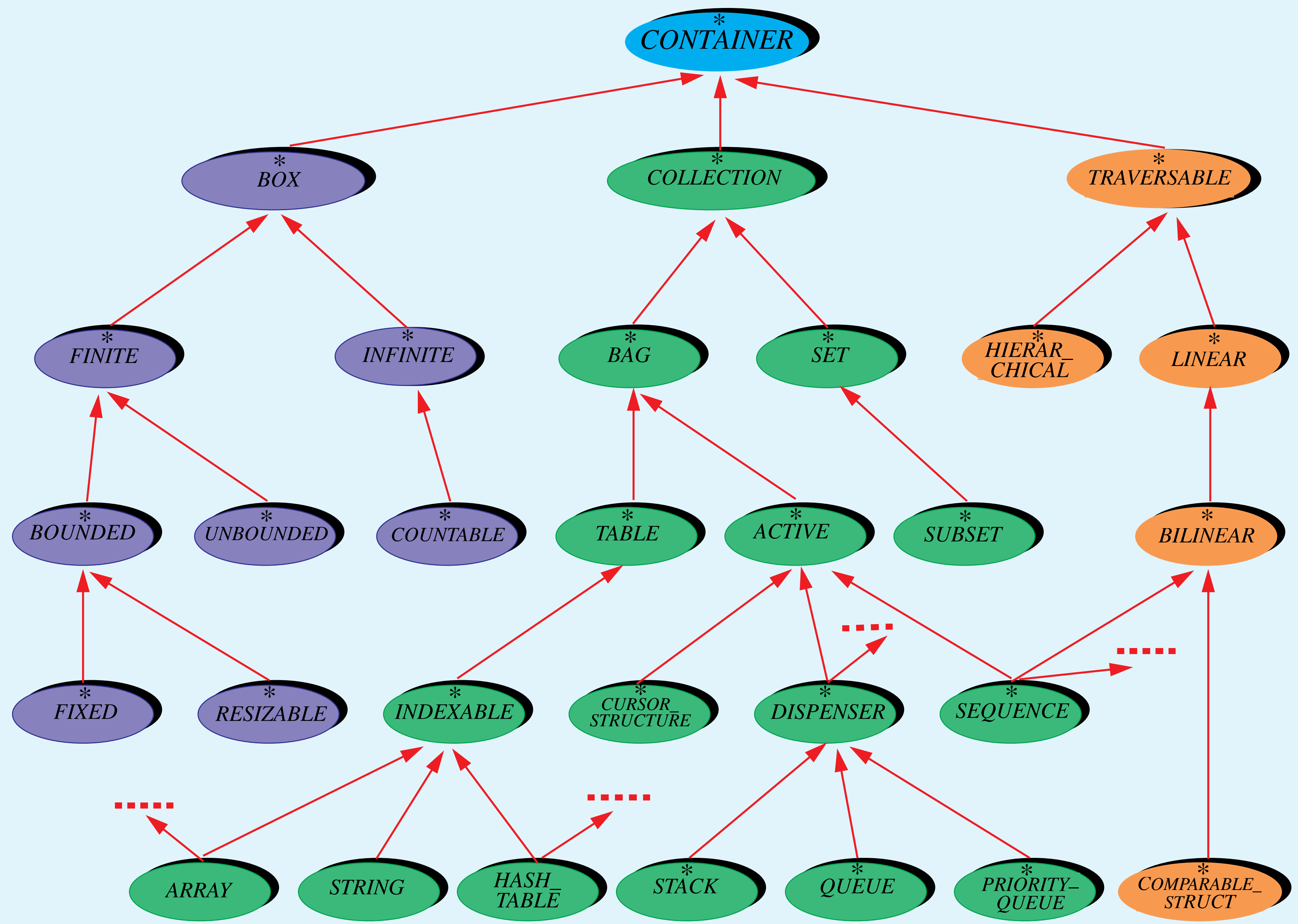
EIFFELCASE

Analysis and design ("upper-case") workbench based on the BON method.

AN ENVIRONMENT FOR O-O DEVELOPMENT

- **Openness: easily interfaced with existing software, other tools. Support for both out and in calls.**
- **Support for cross-development: C package generation.**
- **Garbage collection.**
- **Automatic compilation management.**
- **Very fast incremental reexecution.**
- **Browsing facilities.**
- **Graphical analysis and design facilities.**
- **Source debugging.**
- **Automatic documentation facilities: *short, flat, flat-short*....**
- **Relational database interfaces.**
- **Libraries of robust reusable components.**

CONTAINER STRUCTURES



COMPLEX FIGURES (Continued)

Note: a simpler form of procedures *display*, *move* etc. can be obtained through the use of iterators. The resulting form is just:

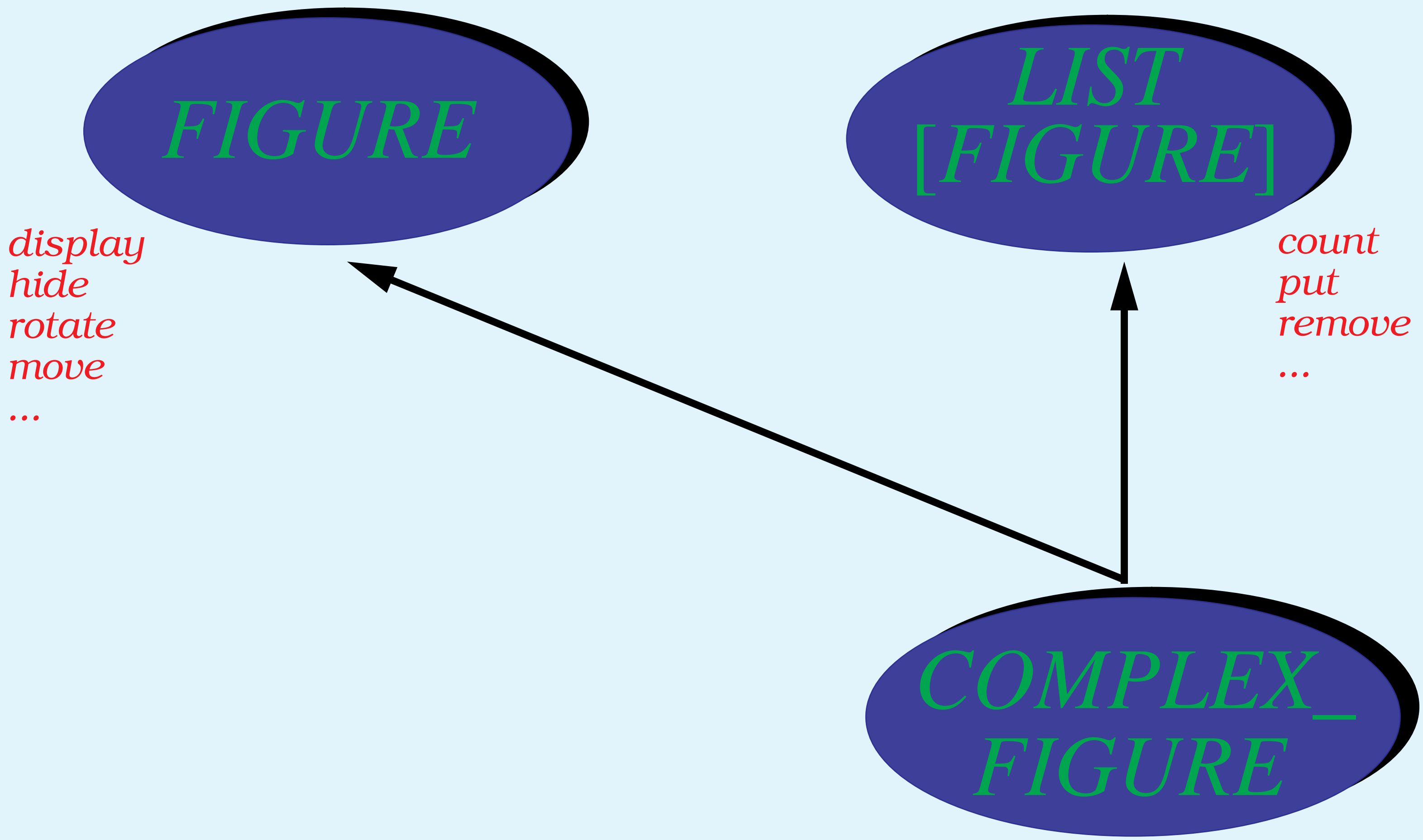
```
display is
    do
        displayer.iterate
    end;

move (x, y: REAL) is
    do
        mover.set_move (x, y);
        mover.iterate
    end;

etc.
```

COMPLEX FIGURES

```
class COMPLEX_FIGURE inherit
   FIGURE
      redefine display, move, rotate, ... end;
   LIST [FIGURE]
feature
   display is
         -- Display each constituent figure in turn.
      do
         from
            start
         until
            after
         loop
            item.display;
            forth
         end
      end;
   ... Similarly for move, rotate etc. ...
end
```

DEFINING THE NOTION OF COMPLEX FIGURE

THROUGH MULTIPLE INHERITANCE



TECHNIQUES (continued)

Genericity:

- **Unconstrained.**

class *STACK* **[***G***]** ***feature*** **...**

- **Constrained.**

class *VECTOR* **[***G* **–>** *NUMERIC***]** ***feature...***

Once routines.

Obsolete classes and features.

Assignment attempt.

```
x ?= y;

if x /= Void then
    ...
    x.some_feature;
    ...
end
```

TECHNIQUES (continued)

More inheritance techniques

Deferred classes and features.

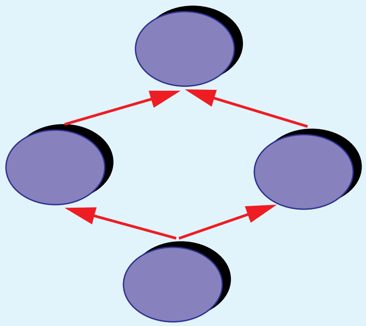
Multiple inheritance.

Renaming:

- **Multiple inheritance: resolving name clashes.**
- **Local adaptation of names.**

Repeated inheritance:

- **Sharing *vs.* replication.**
- **Conflicting redeclarations: *select*.**
- **Keeping the inherited version of a redeclared feature.**



TYPING, POLYMORPHISM AND DYNAMIC BINDING

Assume:

p: POLYGON; r: RECTANGLE; t: TRIANGLE; x: REAL;

Permitted:

x := p.perimeter; x := r.perimeter;
x := r.diagonal;

p := r

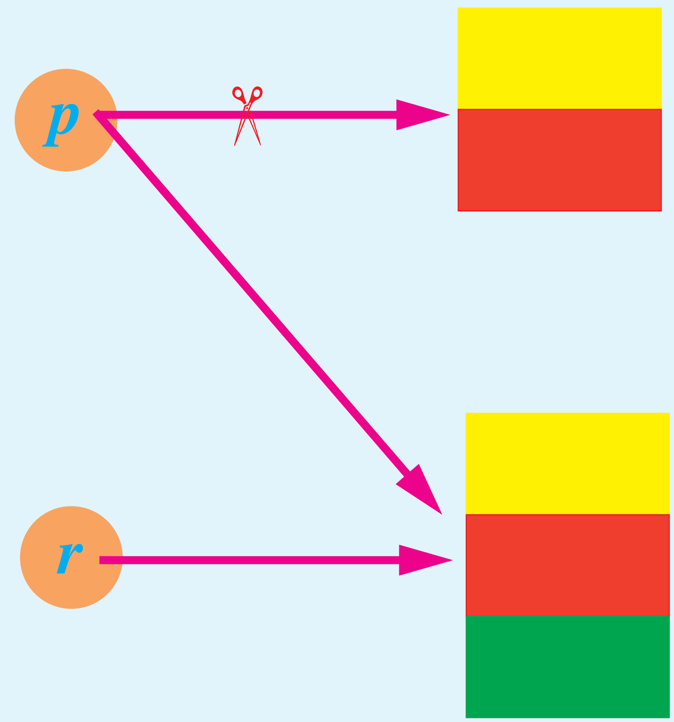
NOT permitted:

x := p.diagonal;

r := p

What is the effect of the following (assuming *some_test* is true)?

```
if some_test then
    p := r
else
    p := t
end;

x := p.perimeter
```



POLYMORPHISM: *p* **may have different forms at run-time.**

DYNAMIC BINDING: Effect of *p.perimeter* **depends on run-time form of** *p***.**

Inheritance techniques

Single inheritance.

Redefinition.

Polymorphism.

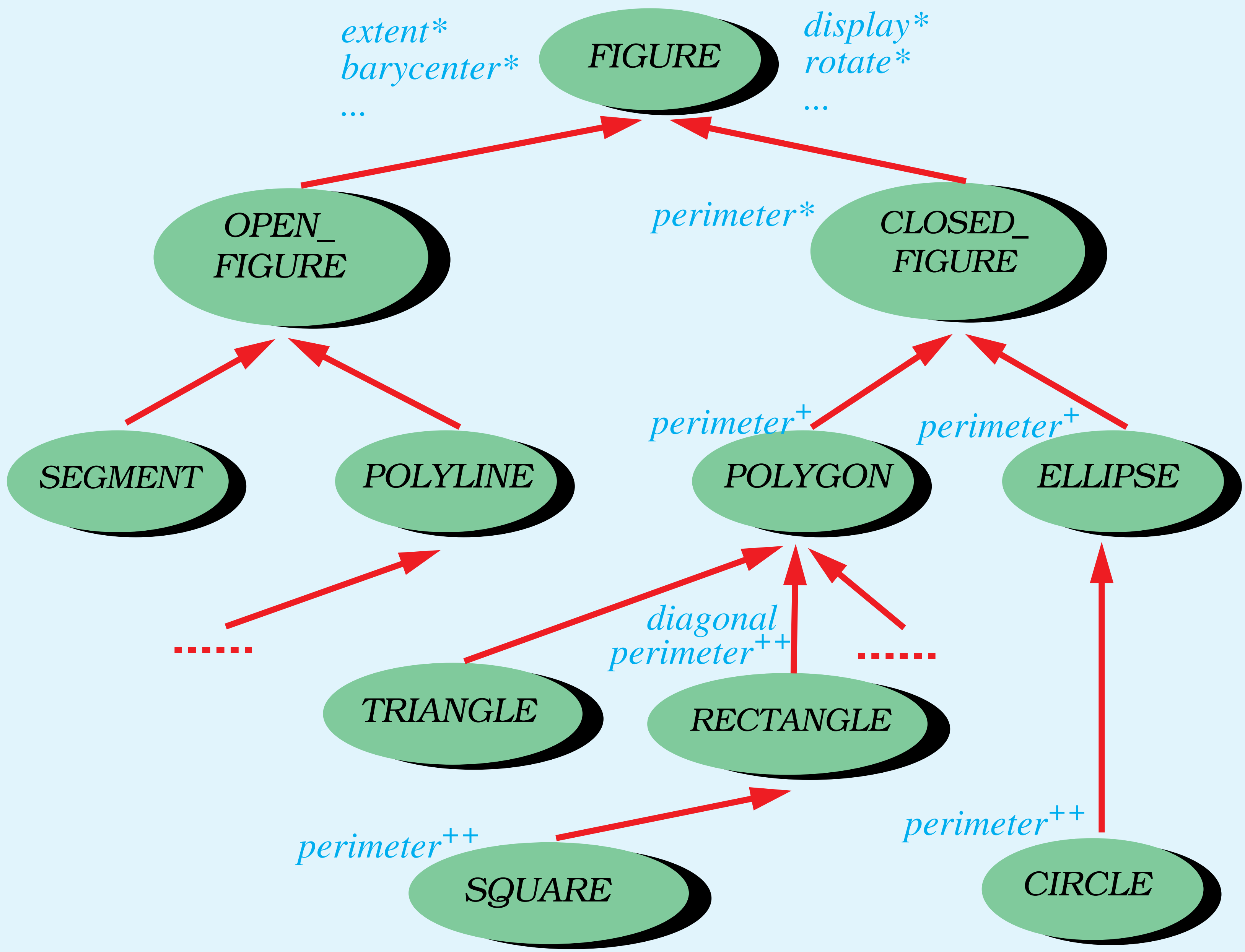
Static typing + Dynamic binding.

$x \bullet f\,(\ldots)$

e.g.
my_plane•lower_landing_gear

- **Static typing: *at least one* feature**
- **Dynamic binding: the *right* feature**

INHERITANCE



DESIGN BY CONTRACT

Applications:

- **Analysis and Design.**
- **Built-in reliability.**
- **Controlling inheritance (sub-contracting).**
- **Basis for exception handling.**
- **A systematic method for testing, debugging, quality assurance.**

See: *Applying "Design by Contract"*, B. Meyer, IEEE Computer, November 1992.

WHAT ARE ASSERTIONS GOOD FOR?

- Writing correct software (analysis, design, implementation, maintenance).
- Documentation
  (the "short" form of a class)

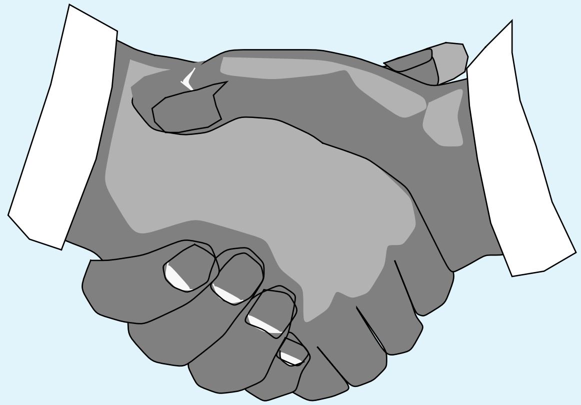
- -

- Quality assurance, testing, debugging.
  (especially in connection with the use of libraries)
- Exception handling

Compilation options (per class):

- No assertion checking
- Preconditions only
- Preconditions and postconditions
- Preconditions, postconditions, class invariants
- All assertions

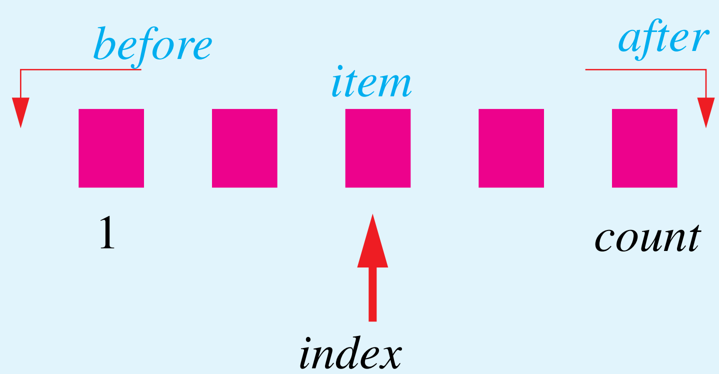
DESIGN BY CONTRACT



put_right	**OBLIGATIONS**	**BENEFITS**
Client	*(Satisfy precondition:)* **Make sure cursor is at valid insertion position.**	*(From postcondition:)* **Get *x* added to the list at desired position.**
Supplier	*(Satisfy postcondition:)* **Insert *x* to right of cursor position; bring cursor to new item.**	*(From precondition:)* **Simpler processing thanks to assumption that cursor is valid insertion position.**

ASSERTIONS!

```
class ARRAYED_LIST [G] inherit
    LIST
            redefine put_left, put_right end;
    ARRAY
            rename put as array_put end
feature
    put_right (x: G) is
            -- Insert x right of cursor position; move cursor to new item.
        require
            not after
        do
            ...
        ensure
            item = x;
            count = old count + 1;
            index = old index + 1
        end;
    ... Other features ...
    invariant
            0 <= index; index <= count + 1
end
```



```
| | Subject: AT&T Bug
| | Date: Fri Jan 19 12:18:33 1990
| |
| | This is the bug that caused the AT&T breakdown
| | the other day:
| |
| | In the switching software (written in C),
| | there was a long "do . . . while" construct,
| |     which contained a "switch" statement,
| |         which contained an "if" clause,
| |             which contained a "break",
| |                 which was intended for the "if" clause,
| |                     but instead broke from the "switch".
| |
```

SIMPLICITY

Some of what is NOT in Eiffel:

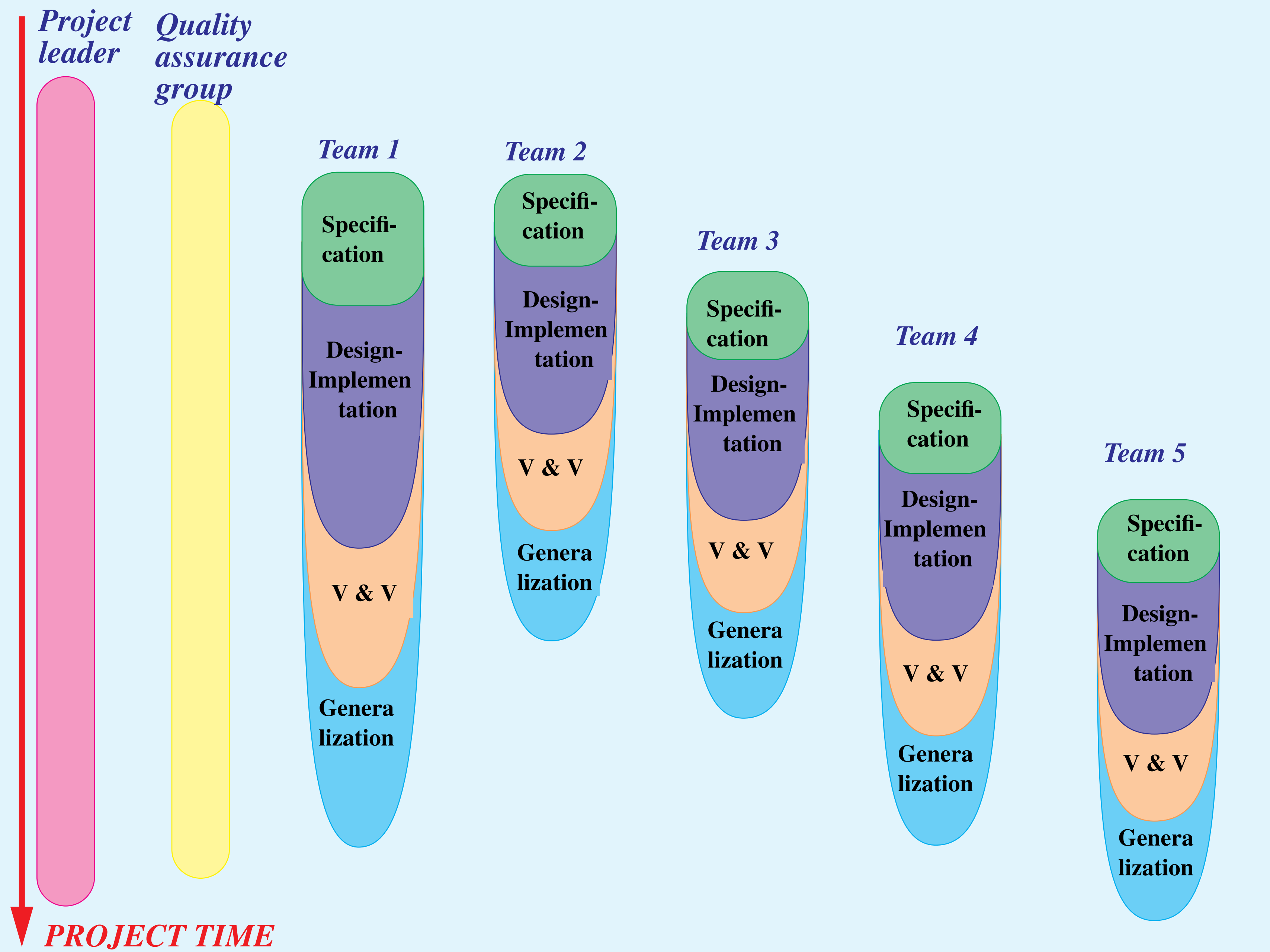
- **Global variables.**
- **Pointer arithmetic.**
- **Goto.**
- **Exit, return etc.**
- **For loops, repeat loops etc.**
- **Main program.**
- **Routines as arguments.**
- **Union types (or records with variants etc.).**
- **Operators with side effects.**

[C]	*y* = *x*++	[5 keystrokes]
[Eiffel]	*y* := *x*; *x* := *x*+*1*	[11 keystrokes]

Combination of facilities:

- CLASSES (Module ≡ Type)
- MULTIPLE INHERITANCE
  Renaming, redefinition, "join"...
- STATIC TYPING
- DYNAMIC BINDING
- GENERICITY
- ASSERTIONS
- EXCEPTION HANDLING
- DEFERRED CLASSES
- GARBAGE COLLECTION
- AUTOMATIC RECOMPILATION
- SOURCE-LEVEL DEBUGGING
- *** STANDARD LIBRARIES ***
- AUTOMATIC PERSISTENCE MECHANISM

THE CLUSTER LIFECYCLE



A COROLLARY OF SEAMLESSNESS:

REVERSIBILITY

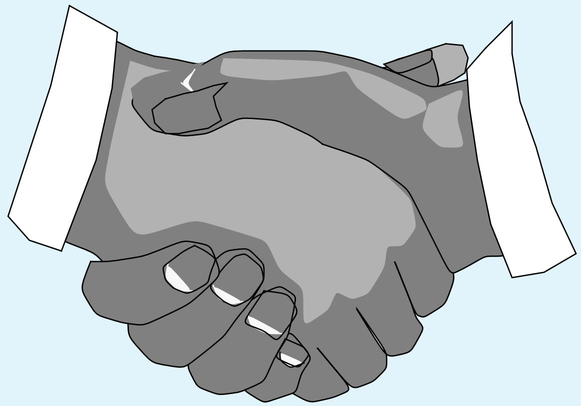
It must be possible to update the implementation, design and analysis continuously.

See:

- ***Seamless Object-Oriented Software Architectures: Analysis and Design of Reliable Systems***

  Kim Waldén, Jean-Marc Nerson, Prentice Hall, 1994.

CONTRACTS FOR ANALYSIS



fill	OBLIGATIONS	BENEFITS
Client	(*Satisfy precondition:*) Make sure input valve is open, output valve is closed.	(*From postcondition:*) Get filled-up vat, with both valves closed.
Supplier	(*Satisfy postcondition:*) Fill the vat and close both valves.	(*From precondition:*) Simpler processing thanks to assumption that valves are in the proper initial position.

OBJECT-ORIENTED ANALYSIS: USING CONTRACTS AND DEFERRED CLASSES

```
deferred class VAT inherit
        TANK
feature
    fill is
            -- Fill the vat.
        require
            in_valve.open; out_valve.closed
        deferred
        ensure
            in_valve.closed; out_valve.closed; is_full
        end;

    is_full, is_empty, empty, in_valve, out_valve, gauge,
    maximum, ... [Other features] ...

invariant
    is_full = (gauge >= .97 * maximum) and
              (gauge <= 1.03 * maximum)
end
```

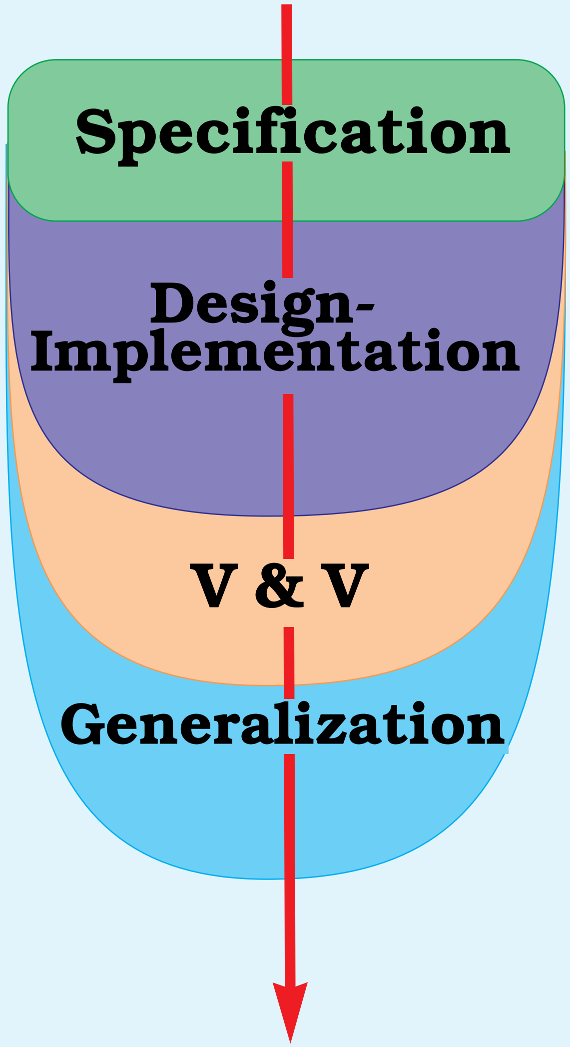
Precondition

-- i.e. specified only,
-- not implemented.

Postcondition

Class invariant

SEAMLESS DEVELOPMENT



- *TRANSACTION, PLANE, CUSTOMER, ENGINE…*
- *STATE, USER_COMMAND, HASH_TABLE, LINKED_LIST…*
- *TEST_DRIVER, …*
- *LIST, …*

Example classes

Generalization

- *LIST, ...*

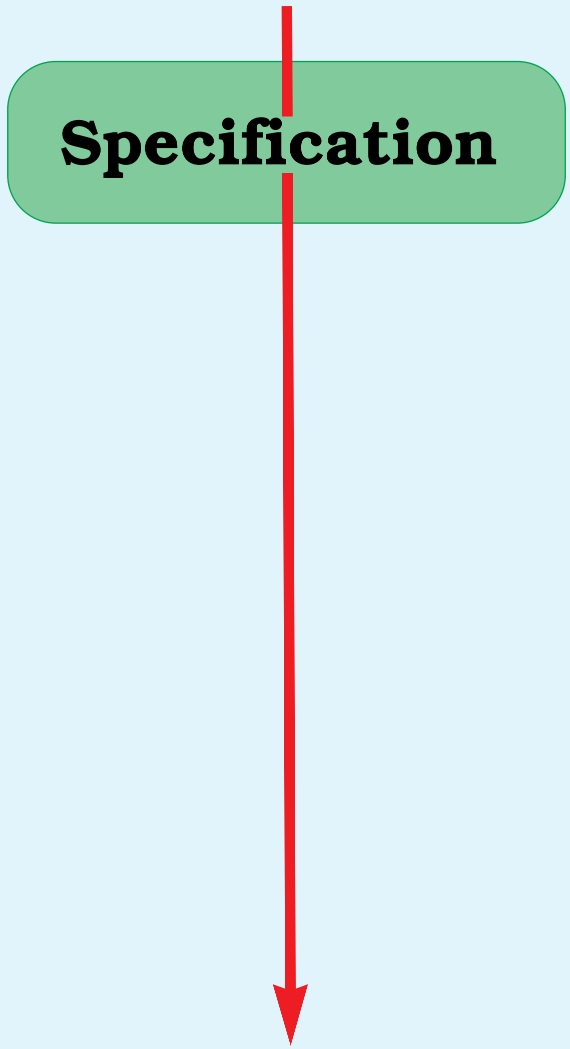
V & V

- ***TEST_DRIVER, ...***

Design-Implementation

- ***STATE, USER_COMMAND, HASH_TABLE, LINKED_LIST...***

SEAMLESS DEVELOPMENT



- *TRANSACTION, PLANE, CUSTOMER, ENGINE...*

Example classes

SEAMLESS DEVELOPMENT

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

Each step enriches the previous one in two ways:

- Add new classes.
- Complete previous classes.

Major advantages:

- Smooth process. Avoids gaps (improves productivity, reliability).
- The *DIRECT MAPPING* property: mapping from problem to solution, i.e. from software system to external model.
- Better traceability of solution to problem.
- Better responsiveness to customer requests.
- Consistency, ease of communication.
- Better interaction between users, managers and developers.

THE PRODUCT CULTURE

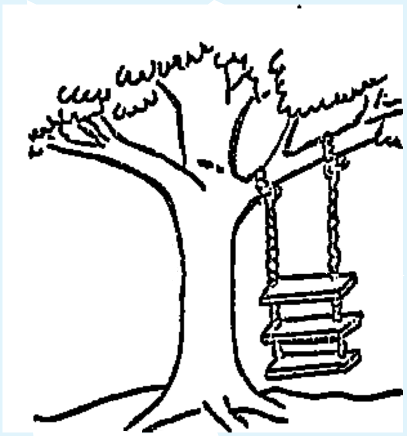
- OUTCOME: Tools, libraries, components
- UNIT: Department, company, industry...
- ECONOMICS: Investment
- TIME: Long-term
- GOAL: System
- BRICKS: Software Components
- STRATEGY: Bottom-up
- METHOD: Object-oriented
- LANGUAGE: Object-oriented

THE ESCHERFALL

M.C Escher:
Waterval

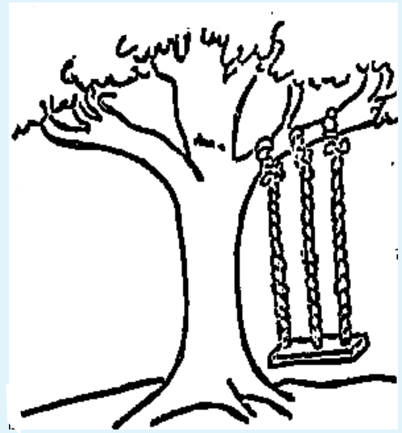
IMPEDANCE MISMATCHES

1



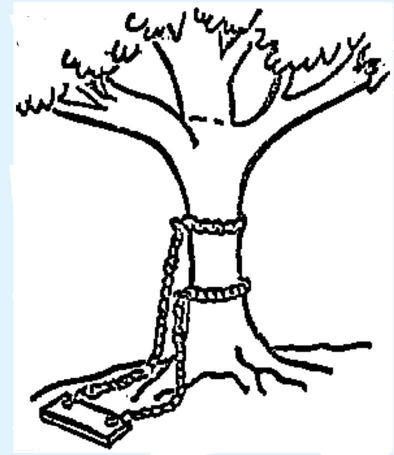
As Management requested it.

2



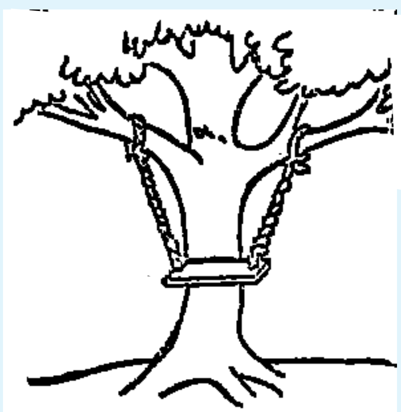
As the Project Leader defined it.

3



As Systems designed it.

4

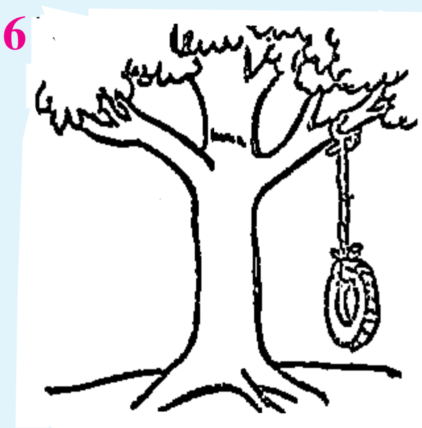


As Programming developed it.

5

As Operations installed it.

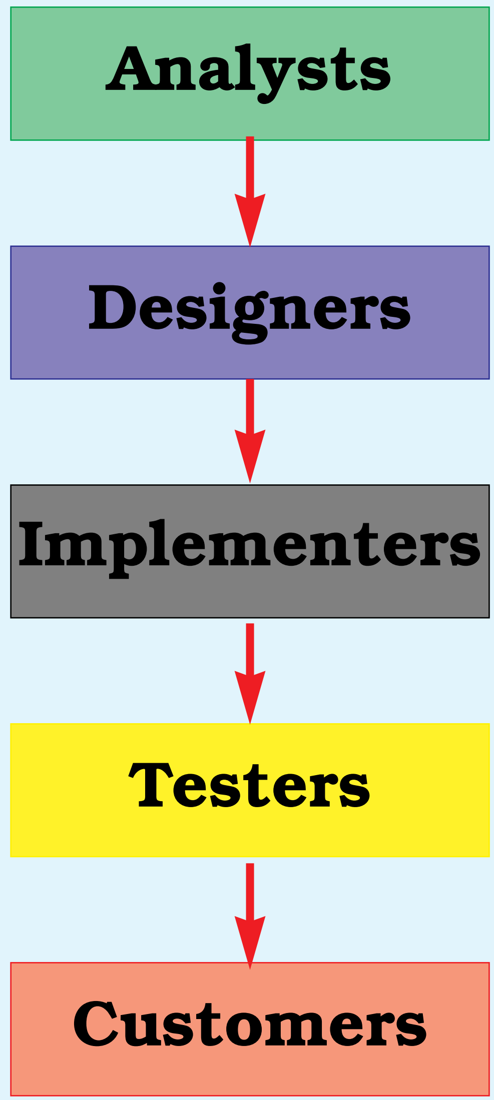
6



What the user wanted.

(Pre-1970 cartoon; origin unknown)

QUALITY CONTROL IN THE TRADITIONAL APPROACH?



PROBLEMS WITH THE WATERFALL

Late appearance of actual code.

Lack of support for requirements change — and more generally for extendiblity and reusability.

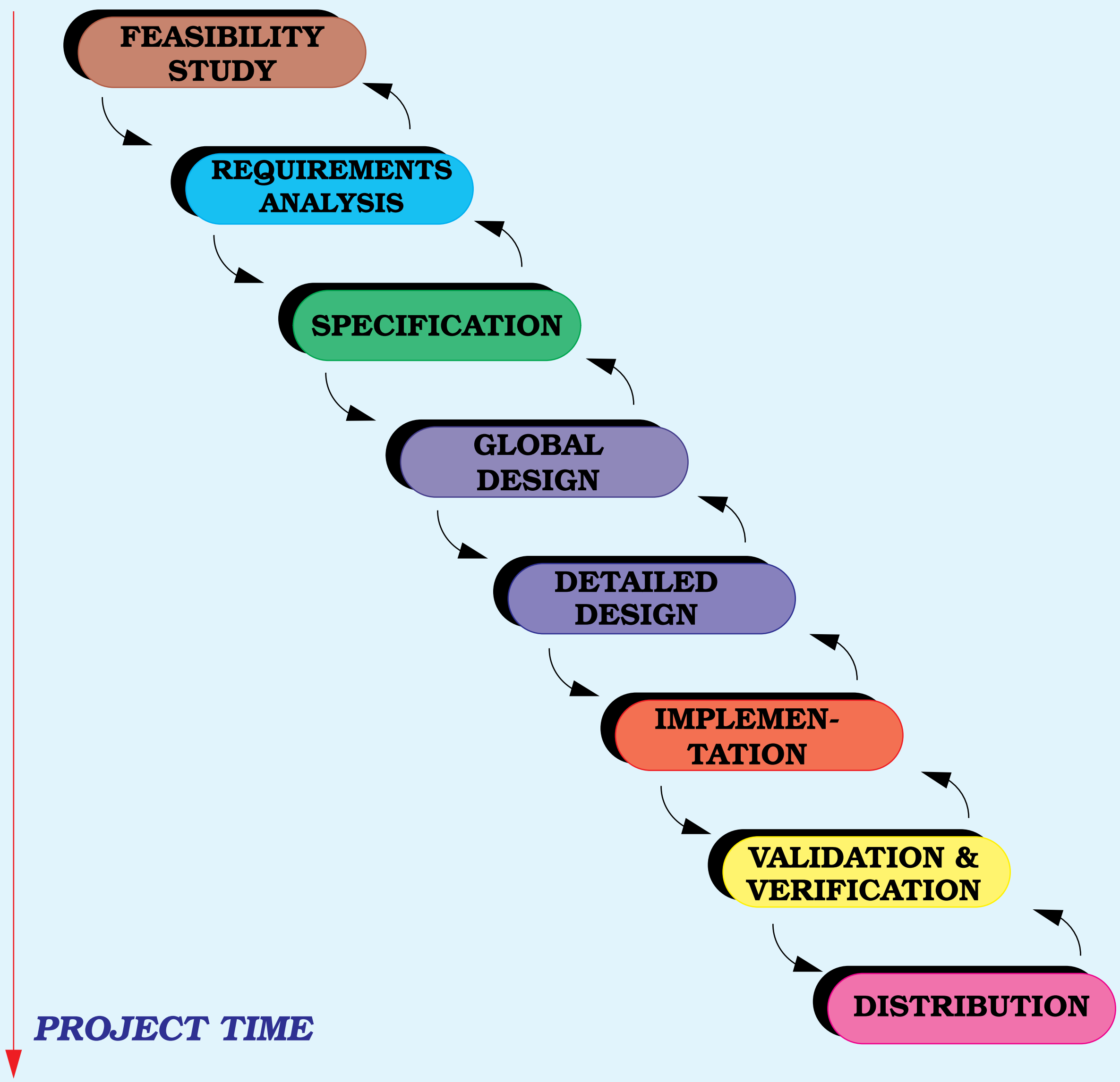
Lack of support for the maintenance activity (70% of software costs?)

Division of labor hampering Total Quality Management.

Impedance mismatches.

Highly synchronous model.

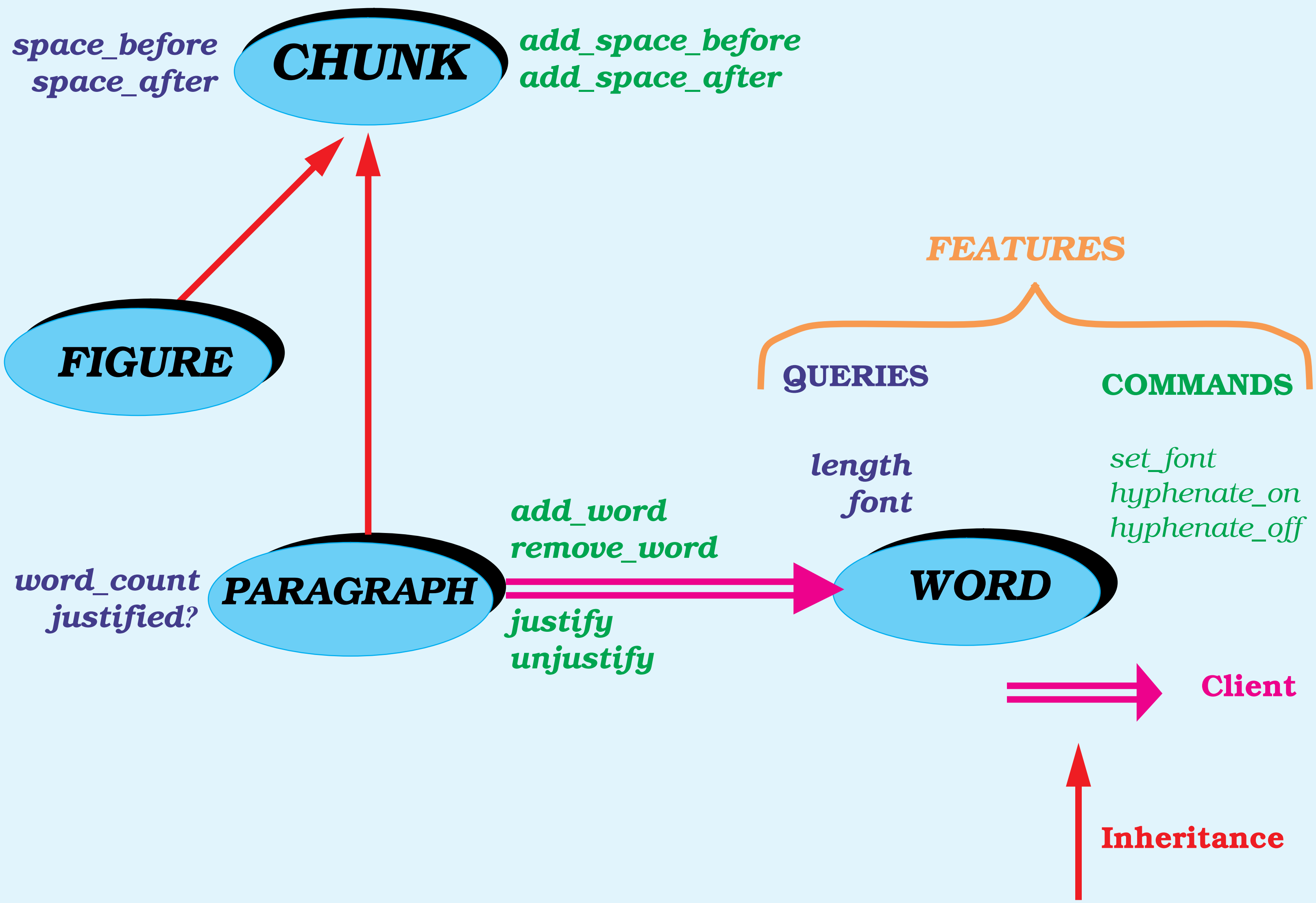
THE WATERFALL MODEL OF THE SOFTWARE LIFECYCLE



THE PROJECT CULTURE

- **OUTCOME: Results**
- **UNIT: Project**
- **ECONOMICS: Profit**
- **TIME: Short-term**
- **GOAL: Program**
- **BRICKS: Program elements**
- **STRATEGY: Top-down**
- **METHOD: Functional, structured analysis/design, ...**
- **LANGUAGE: C, Pascal, ...**

OVERALL SYSTEM STRUCTURE



KEY IDEAS

- System decomposition and architecturing technique. "Programming-in-the-small" aspects less crucial.
- Flexible, decentralized software architectures. No "center" or "top".
- Software construction by bottom-up combination and extension of reusable software components.
- Partially refined components, capturing common behaviors.
- Precise specification of individual components: assertions.
- Other reliability techniques: static typing, exception handling.
- Systematic approach to the classification and cataloging of software components.
- Openness to the external world (external routines, packaging of existing tools, support for cross-development).
- Rich development environment (fast recompilation, documentation, browsing, debugging, user interfaces).
- LIBRARIES.

WHAT IS OBJECT TECHNOLOGY?

- **An architectural principle:**

  MODULE = TYPE **(CLASS)**

- **An epistemological discipline:**

  ABSTRACTION

- **A classification rule:**

  INHERITANCE

- **A correctness requirement:**

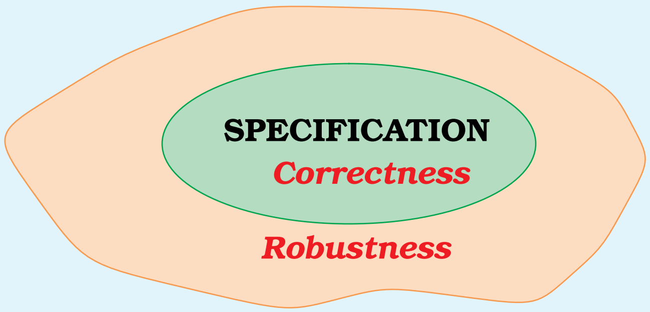
  DESIGN BY CONTRACT

- **An engineering obligation:**

  REUSABILITY AND EXTENDIBILITY

THE GOAL: SOFTWARE QUALITY AND PRODUCTIVITY

- **REUSABILITY.**
- **EXTENDIBILITY.**
- **RELIABILITY (Correctness + Robustness).**
- **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.

"When quality is pursued, productivity will follow" (K. Fujino, VP, NEC Corporation, quoted in *Software Engineering* by C. Ghezzi et al.)

COVERAGE (ISE Eiffel)

Full lifecycle coverage

- Analysis
- Design
- Implementation
- Maintenance

Full application coverage

- Graphics
- User Interfaces
- Databases
- Networking

Broad platform coverage

- Windows 3.1, Windows NT, Windows 95
- Unix: OSF/1, SunOS, Solaris, AIX, HP, SCO, Linux, DG, ...
- VMS.
- NEXTSTEP.
- In progress: OS/2, MacOS

IS THIS IT?

```
// These classes implement the necessary logic
// for building reference counted objects
// and the associated pointers.

class counted {
        friend class Rptr;
        int nreferences;
                // number of references to
                // this object
        addRef ()
                { nreferences++; }
        delRef ()
                {
                        if (--nreferences)
                                delete this;
                }
  public:
        Counted ()
                { nreferences = 0; }
        ~Counted();
} ;

Class RPtr_base {
  protected:
        Counted *ptr;
        RPtr_base()
                { ptr = 0; ]
        RPtr(RPtr& r)
                { ptr = r.ptr;
                 if(ptr) ptr–>addref();
                 return *this;
                }
        RPtr(Counted *tp)
                { ptr = tp;
                if(ptr) ptr–>addref();
                return *this;
                }
        ~RPtr()
                { if (ptr)  ptr–>delref(); }

        RPtr& operator=(RPtr& r)
                { if (ptr) ptr–delref();
                 ptr = r.ptr;
                 if(ptr) ptr–>addref();
                 return *this
                }
        RPtr& operator=(Counted *tp)
                { if (ptr) ptr–>delref();
                 ptr = tp;
                 if(ptr) ptr–>addref();
                 return *this
                }

#define RPtr (T) name2 (Rptr_,T)

#defineMakeRPtr (T)
class RPtr (t) : public RPtr_base {
                Counted *ptr;
  public:
        RPtr(T) ()
                : RPtr_base()
                { }
        RPtr(T) (RPtr(T)& r)
                : RPtr_base(r)
                { }
        RPtr(T) (T *tp)
                : RPtr_base ((Counted *) T)
                { }
        ~RPtr (T) ()
                { }
        RPtr(T)& operator=(RPtr(T)& r)
                { *((RPtr_base *) this) = r; }
        RPtr(T)& operator=(T *tp)
                { *((RPtr_base *) this) =
                        (Counted *) tp; }

        T& operator *()
                { assert(ptr); return *((T *) ptr); }

        operator T *()
                { return (T *) ptr; }
        int operator !()
                { return !ptr; }
};
```

***Source*:**

Jonathan S. Shapiro:

A C++ Toolkit

Prentice Hall Series in Innovative Technology, 1991, pp. 222-223.

OBJECT-ORIENTED SOFTWARE EDUCATION

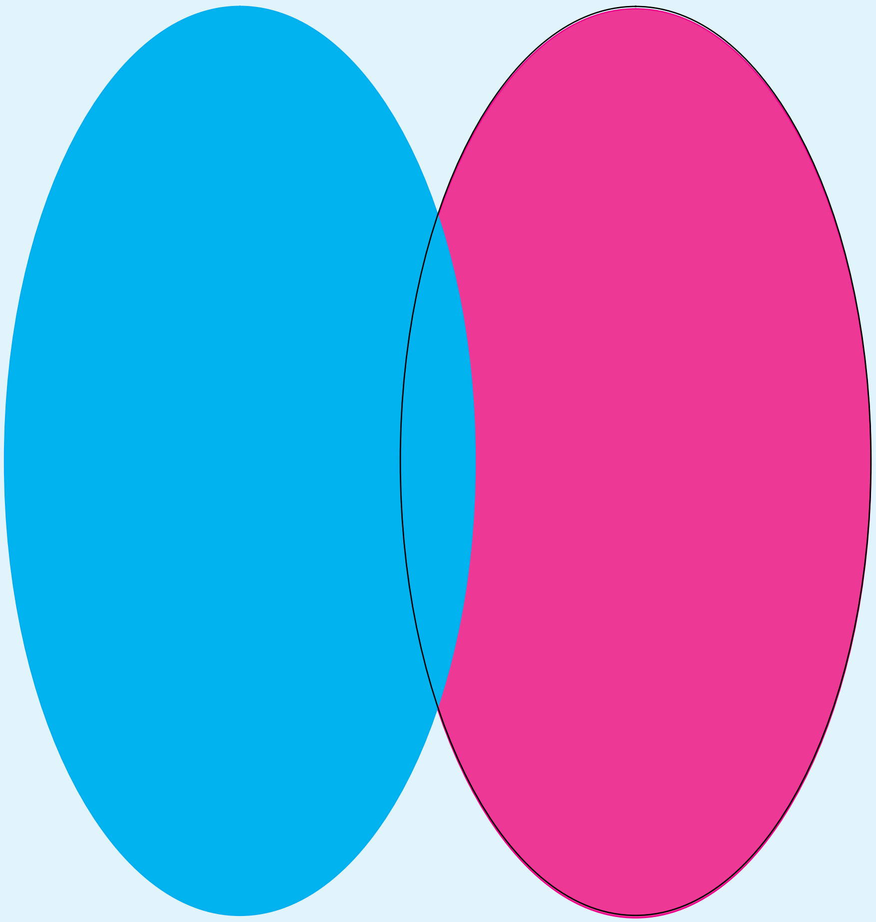
Eiffel is an ideal vehicle for teaching programming and software topics at all levels.

- **Assertions, design by contract.**
- **Gentle introduction to formal specifications.**
- **Inheritance.**
- **Libraries.**
- **Stepwise opening of the black boxes.**
- **Multi-year strategy.**

Eiffel is also an excellent vehicle for teaching introductory programming. Examples: Rochester Institute of Technology, University of Technology Sydney, University of Nice, Technische Universität Darmstadt, Rochester Institute of Technology, Hartford Graduate Center, ...

See: *Towards an Object-Oriented Curriculum*, B. Meyer, Journal of Object-Oriented Programming, May 1993. Also: *Eiffel in der Schule und Hochschule* conference, Darmstadt, May 1995.

EIFFEL'S ORIGINALITY



SCIENTIFICALLY RESPECTABLE

USEFUL

SAMPLE PROJECTS

Bank of New York (USA): Message switching system.

Tetra Ltd. (UK): On-Line Transaction Processing System.

ESPRIT Programme (EEC, 7 countries): Business Class project.

Telecom Australia: Protocol analysis and verification.

French Geological Survey (BRGM): Geographical applications.

Appel & Frenzel GmbH (Germany): Business applications.

Bytex (USA): LAN software.

Société Générale (France): Option trading system.

CALFP (Crédit Agricole Lazard Financial Products, UK/USA/France): Derivatives trading system.

RIEM (USA): Investment banking.

SITES (ISE Eiffel)

As of late 1994: more than 5000 sites.

Distribution:

- **60% industrial, 40% academic.**
- **40% US, 40% Europe, 10% Japan, 5% Australia/NZ**

Areas of application:

- **Banking, finance.**
- **Telecommunications.**
- **Software engineering, CASE.**
- **Decision support systems.**
- **Health care.**
- **Education.**

TOOLS CONFERENCES

Technology of Object-Oriented Languages and Systems.

The major forum for presenting and discussing applications of object-oriented technology. Rich tutorial program; workshops; fair representation of all trends in the object-oriented field.

International conference organized since 1989, on three continents.

Previous locations: Paris, Santa Barbara, Sydney, Dortmund.

Proceedings published by Prentice Hall.

Next conference: TOOLS USA '95, Santa Barbara, 29 July-August 2. Chairman is Raimund Ege from Florida International University.

Eiffel is an Open System: THE NONPROFIT INTERNATIONAL CONSORTIUM FOR EIFFEL (NICE)

The institution responsible for controlling Eiffel and its evolution

- **Nonprofit**
- **International base (US, Canada, Europe, Australia). Members include Telecom Australia, Reuters, Bytex, ENEA, GEC-Marconi etc.**
- **Works through technical committees: languages, libraries, future directions.**
- **Established: October 1990**
- **Membership (Regular or Charter) open to any seriously interested company. Also: individual members.**
- **Also: educational associates (universities).**
- **Most recent achievement: the ELKS (*Eiffel Library Kernel Standard*), Vintage 1995.**

EIFFEL TODAY

Non-proprietary base technology:

- **Language**
- **Kernel library**

NICE (Nonprofit International Consortium for Eiffel).

TEN FREQUENT QUESTIONS ABOUT EIFFEL

- Is it public-domain?
- Does it run on PCs?
- Is it compatible with C software?
- Does it support graphics?
- Is there an interface builder?
- Is there a standardized library?
- Are there several providers of reusable components?
- Does it support an object-oriented analysis method?
- Is it a programming language, a method, or a development environment?
- Why is it called that way?

RELATED DOCUMENTATION

- *OBJECT SUCCESS: A Manager's Guide to Object Technology, its Impact on the Corporation, and its Use for Reengineering the Software Process*, B. Meyer, Prentice Hall, 1995. A presentation for decision-makers.
- *Object-Oriented Programming in Eiffel*, Robert Rist and Robert Terwilliger, Prentice Hall, 1995. Introductory programming textbook based on Eiffel.
- *Object-Oriented Programming in Eiffel*, Pete Thomas and Ray Weedon, 1995. Another introductory programming text.
- *Software Development Using Eiffel: There Can be Life After C++*, Richard Wiener, Prentice Hall, 1995.
- *Introduction to the Theory of Programming Languages*, B. Meyer, Prentice Hall, 1990. Provides some of the theoretical basis.

Several textbooks in preparation. Addison-Wesley has started an Eiffel series.

DOCUMENTATION

- *Object-Oriented Software Construction*, B. Meyer, Prentice Hall, 1988. Presents the method. (Translations: Italian, Japanese, French, German, Dutch, Spanish, Chinese.)
- *Object-Oriented Applications*, eds. B. Meyer and J.-M. Nerson, Prentice Hall, 1993.
- *Eiffel: The Language*, Prentice Hall, 1991. Complete language description.
- *Reusable Software: The Base Object-Oriented Libraries*, Prentice Hall, 1994.
- *An Object-Oriented Environment: Principles and Applications*, Prentice Hall, 1994.
- *Seamless Object-Oriented Architecture: Analysis and Design of Reliable Systems*, Kim Waldén &Jean-Marc Nerson. Prentice Hall, 1994. (The conceptual basis for BON and EiffelCase.)
- *Eiffel: An Introduction*, Robert Switzer, Prentice Hall, 1993.
- *An Eiffel Collection*. Collection of relevant articles.

EIFFEL BACKGROUND (Continued)

Spring 1990: Documents describing new language level, Eiffel 3, are made available following extensive discussions on the Net.

October 1990: NICE (Nonprofit International Consortium for Eiffel) established.

1991: First printing of *Eiffel: The Language* published, describing Eiffel 3. (Current printing is from 1992.)

September 1993: Release of ISE Eiffel 3 (version 3.1).

May 1995: Release of ISE Eiffel 3.3, including graphical version for Windows (personal and professional).

EIFFEL BACKGROUND

1973-1985: Experience with Simula 67, building reusable software, teaching modern software techniques in an industrial environment, abstract data types, scientific software, formal specification languages (Z, M), Ada, teaching data structures and algorithms.

September 1985: Eiffel designed.

October 1986: Implementation demonstrated at first OOPSLA conference. Paper on *Genericity vs. Inheritance*.

March 1988: Publication of *Object-Oriented Software Construction* by Prentice Hall.

December 1988: *comp.lang.eiffel* Usenet newsgroup created.

PLAN

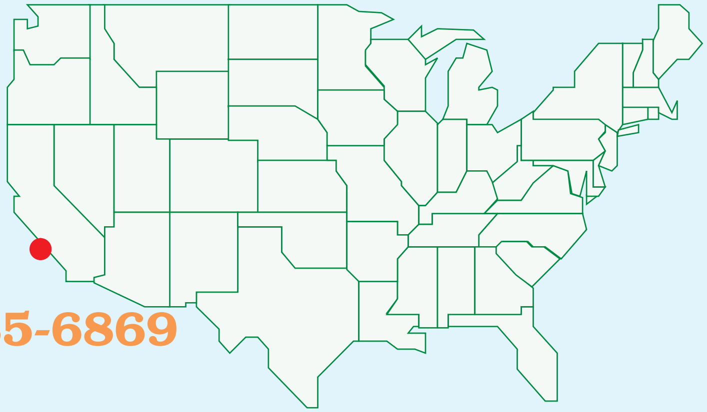
1. ISE and Eiffel: background.
2. The Eiffel method.
3. The structure of the environment
4. Conclusion – questions

EIFFEL:

AN OVERVIEW

Bertrand Meyer

ISE Inc.



Author's address:

Interactive Software Engineering Inc.
270 Storke Road, Suite 7
Santa Barbara, CA 93117 USA
Telephone 805-685-1006, Fax 805-685-6869
E-mail <info@eiffel.com>

WWW: http://www.eiffel.com — Anonymous FTP: eiffel.com