AT THE EDGE OF
DESIGN BY CONTRACT

Bertrand Meyer
Interactive Software Engineering

http://www.eiffel.com

TOOLS EUROPE, Zürich, 12 March 2001

PLAN

1. Design by Contract: background and scope

2. Issues to which I don’t know the solution

DESIGN BY CONTRACT

Confluence of work from:

- Axiomatic semantics of programming (Hoare 1969-1972)
- “Proof of correctness of Data Representations” (Hoare 1972)
- “Constructive approach” (Dijkstra 1976)
- Abrial’s Z (197)
- Abstract data types
- Object-oriented programming
- Reuse

THE THREE QUESTIONS

What does it assume?

What does it guarantee

What does it maintain?

THE COST OF NOT ASKING

LOS ANGELES, 9 November 2000. Failure of the Southwest's main air traffic radar system was traced to new software unable to recognize handoff data typed manually by Mexico controllers.

The software installed Wednesday night is the same upgrade completed successfully at 19 other FAA radar centers. But software designers didn't allow for information typed in by Mexico controllers handing off flights.

“The computer didn't recognize the information and it aborted”, a spokesman said. “A digit out of place could do it.”

A CLASS WITH CONTRACTS

```
class WEB_PAGE inherit
    GENERIC_WEB_PAGE
feature
    refresh is
            -- Reload page from server
        require
            valid_connection:  connection.open
        do
            if changed then update end
        ensure
            refreshed: old  changed implies updated
        end
    ...
    changed: BOOLEAN
invariant
    valid_connection: connection /= Void
end -- class WEB_PAGE
```

APPLICATIONS

- **Analysis and design.**
- **Implementation: built-in reliability.**
- **Testing, debugging, quality assurance.**
- **Documentation.**
- **Exception handling.**
- **Inheritance.**
- **Project management: preserving top designers’ work.**

EXAMPLE

Laser printer software at Hewlett-Packard, 1997-1998

About 800,000 lines of legacy C code.

Contracts: first emulated in C/C++ through macros, then Eiffel software added

C calls Eiffel

Some results:

- **Major errors found in the legacy C code.**
- **Bug in chip.**

See eiffel.com

NON-EIFFEL IMPLEMENTATIONS

UML: See OCL tutorial

C++: Macro packages

- Nana (NTU Darwin --> GNU)
- Todd Plessel (Lockheed Martin / EPA)

Java

- iContract
- Biscotti (MITRE)

EMULATING CONTRACTS

Step 1: preconditions and postconditions

Systematic documentation

Next: invariants

Inheritance?

THE CONTRACT WIZARD

Source: ISE

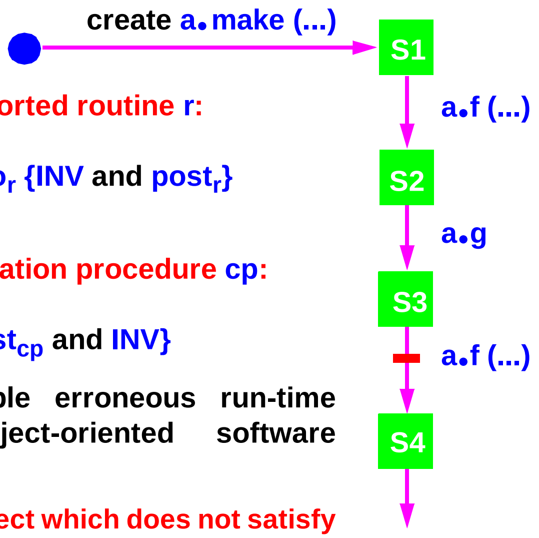
Applicable to Microsoft .NET assemblies

Origin can be any language

User interactively selects classes and routines, and adds preconditions, postconditions and invariants

Wizard produces proxy classes

CLASS CORRECTNESS



(1-*n*) For every exported routine r:

{INV and pre_r} do_r {INV and $post_r$}

(1-*m*) For every creation procedure cp:

{pre_{cp}} do_{cp} {$post_{cp}$ and INV}

The worst possible erroneous run-time situation in object-oriented software development:

- Producing an object which does not satisfy the invariant of its own class.

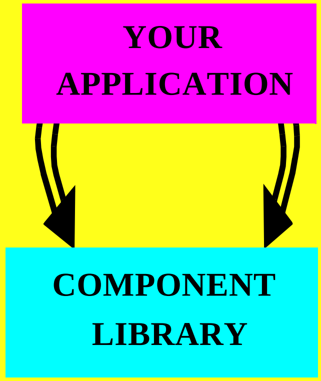
CONTRACTS AND QUALITY ASSURANCE

A run-time assertion violation is the manifestation of a bug:

- **Precondition violation: client bug.**
- **Postcondition or invariant violation: supplier bug.**

CONTRACTS AND BUG TYPES

Preconditions are particularly useful to find bugs in client code:



YOUR APPLICATION

COMPONENT LIBRARY

```
your_list.insert (y, a + b + 1)

class LIST [G] feature
    ...
    insert (x: G; i: INTEGER) is
        require
            i >= 0
            i <= count + 1
```

CONTRACTS AND REUSE

The short form — i.e. the set of contracts governing a class — should be the standard form of library documentation.

Examples:

- **ISE EiffelBench**
- **GEHR**

CONTRACTS AND INHERITANCE

Issues: what happens, under inheritance, to

- **Class invariants?**
- **Routine preconditions and postconditions?**

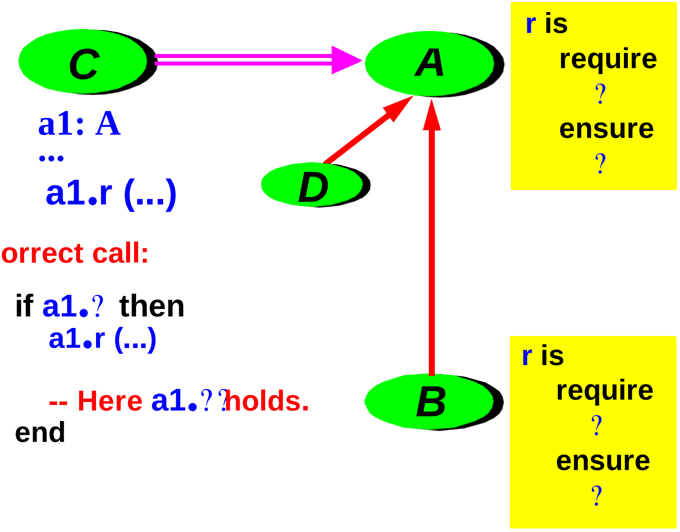
INVARIANTS

Invariant Inheritance rule

The invariant of a class automatically includes the invariant clauses from all its parents, “and”-ed

Accumulated result visible in flat and flat-short forms.

CONTRACTS AND INHERITANCE



r is
require
?
ensure
?

a1: A
...
a1.r (...)

Correct call:

```
if a1.? then
   a1.r (...)

   -- Here a1.? ?holds.
end
```

r is
require
?
ensure
?

ASSERTION REDECLARATION RULE

- **Precondition may only be kept or weakened.**
- **Postcondition may only be kept or strengthened.**

Eiffel rule:Redeclared version may not have require or ensure.

May have nothing (assertions kept by default), or

require else new_pre

ensure then new_post

Resulting assertions are:

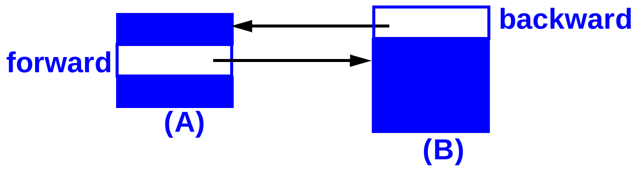
original_precondition or new_pre

original_postcondition and new_post

KNOWN ISSUES: THE INDIRECT INVARIANT EFFECT

Invariant of class A:

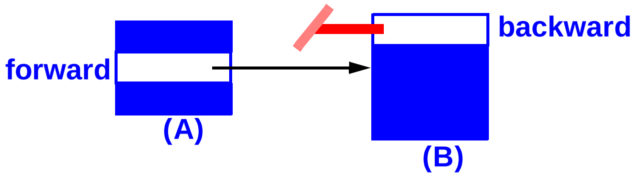
forward.backward = Current



THE INDIRECT INVARIANT EFFECT

Operation in class B:

backward := Void



PROOFS WILL REQUIRE...

... full axiomatization of dynamic aliasing

DESIRABLE MODE OF REASONING

{*SOME_PROPERTY* holds of *a*}

Apply *SOME_OPERATION* to *b*.

{*SOME_PROPERTY* still holds of *a*}

Applicable to “expanded” values, e.g. integers:

{*P* (*a*)}

***OP* (*b*) -- e.g. *b* := *b* + 1**

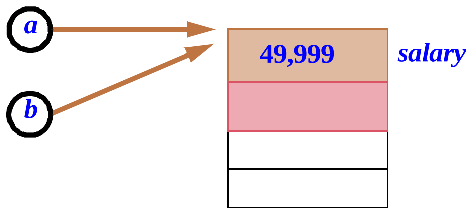
{*P* (*a*)}

REFERENCES CAUSE ALIASING:

{*a* makes less than 50K}

***b*▪*raise_salary* (1)**

{What about *a*?}



NOT JUST IN PROGRAMMING

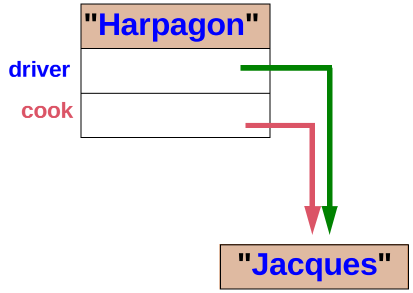
{I heard that one of the CEO’s in-laws makes less than 50K}

Memo to personnel: raise Jill’s salary by one dollar

{?}

METAPHORS ETC.

“Your driver or your cook?”
(to Harpagon)

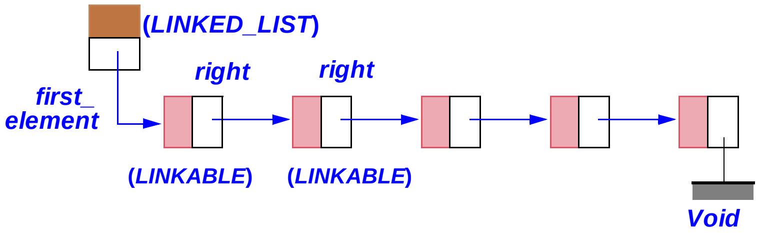


“The beautiful daughter of Leda”

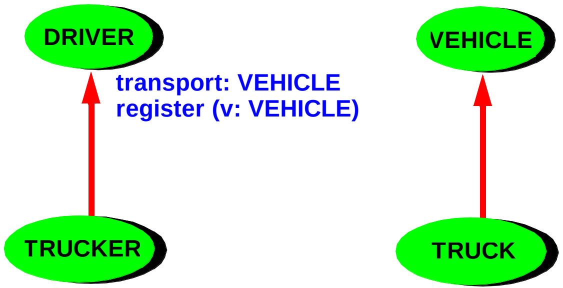
“Menelas’s spouse”

“Paris’s lover”

LINKED LISTS IN EIFFELBASE



COVARIANCE



transport: VEHICLE
register (v: VEHICLE)

transport: TRUCK
register (v: TRUCK)

THE CONTRACT LANGUAGE

How expressive should it be?

Should it permit function calls?

THE CONTRACT LANGUAGE

Language of boolean expressions (plus old):

- No predicate calculus (i.e. no quantifiers, ? or ?).
- Function calls permitted, e.g (in a STACK class):

```
put (x: G) is
        -- Push x on top of stack
    require
        not full
    do
        ...
    ensure
        not empty
    end
```

```
remove is
        -- Pop top of stack
    require
        not empty
    do
        ...
    ensure
        not full
    end
```

EXPRESSING HIGHER-LEVEL PROPERTIES

Use iterators.

Eiffel has **agents**, i.e. routine objects:

```
my_integer_list.for_all (agent is_positive (?))
```

with (in some class)

```
is_positive (x: INTEGER): BOOLEAN is do Result := (x > 0) end
```

or

```
{EMPLOYEE}.for_all (agent is_married)
```

with (in class EMPLOYEE):

```
is_positive (x: INTEGER): BOOLEAN is do Result := (x > 0) end
```

THE IMPERATIVE AND THE APPLICATIVE

do
balance := balance – sum
	ensure
	balance = old balance – sum

PRESCRIPTIVE	DESCRIPTIVE
How	What
Operational	“Denotational”
Implementation	Specification
Instruction	Expression
Imperative	“Applicative”

DIJKSTRA, 1968

“GOTO Statement Considered Harmful”, Comm. ACM

“Our intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.”

FUNCTIONS IN CONTRACTS SHOULD BE “PURE”

No “effects”

Immediately denote mathematical functions

“EFFECT”

Change of state.

The state includes:

- **Set of objects.**
- **Values of their fields (attributes)**
- **State of external devices (e.g. printers)**
- **Values of local variables**

ARE ALL SIDE EFFECTS BAD?

Modify a local variable

```
f: SOME_TYPE is
    local
        x: T
    do
            ... Do something to x ...
            ...
    end
```

ACCEPTABLE SIDE EFFECTS?

Concrete only, no abstract side effect

Complex numbers

Public features:

    add, subtract, multiply, divide, x, y, rho, theta

Secret attributes:

        internal_x, internal_y, internal_rho,
        internal_theta, cartesian_available,
        polar_available, update_cartesian, update_polar

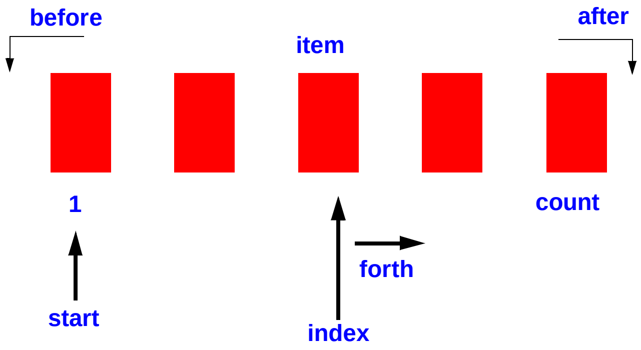
Invariant includes:

            cartesian_available or polar_available

CONCRETE SIDE EFFECT

```
x: REAL is
      -- Abscissa of number
   do
         if not cartesian_available then
            update_cartesian
         end
         Result := internal_x
   end
```

LIST STRUCTURES



before

after

item

1

count

start

forth

index

Implementing the function i_th:

```
    position := index
    go (i)
    Result := item
    go (position)
```

ONCE FUNCTIONS

```
f: SOME_TYPE is

   once

         ... Instructions ...

   end
```

CREATION

```
f: SOME_TYPE is
   do
         create Result.make (...)
   end
```

NEW ENVISIONED EIFFEL CONSTRUCT

```
f is
   require
      ...
   pure
      ...
   ensure
      ...
   end
```

Declaring a routine as "pure" is a proof obligation that it doesn't produce "bad" side effects.

LANGUAGE RULES

A routine is pure if it is side-effect-free or declared as pure.

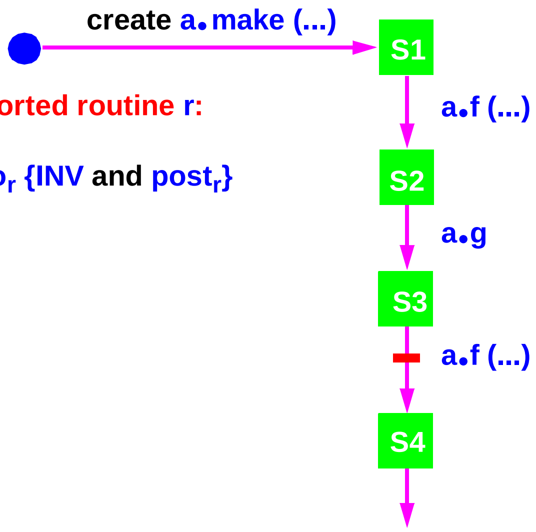
Side-effect free means:

- **No assignment to attributes.**
- **No calls to non-pure routines.**
- **No creations (?).**

Purity must be preserved under redeclaration.

Queries used in assertions must be pure.

THE CALL-IN ISSUE



(1-*n*) For every exported routine r:

$\{INV$ and $pre_r\}$ do_r $\{INV$ and $post_r\}$

UNQUALIFIED VS. QUALIFIED CALLS

Desired properties of calls:

{pre$_r$} r (...) {post$_r$} -- Unqualified

{pre$_r$} x.r (...) {post$_r$} -- Qualified

To be proved:

{pre$_r$} do$_r$ {post$_r$} -- If used in unqualifed calls only

{INV and pre$_r$} do$_r$ {INV and post$_r$}
-- If used in qualifed calls

INVARIANT DOESN’T NEED TO HOLD DURING ROUTINE:

```
r is
        do
            s (...)
                    -- INV not satisfied here
            t (...)
            u (...)
        end
```

WHAT ABOUT:

```
r is
        do
            s (...)
                    -- INV not satisfied here
            x.t (...)
            u (...)
        end
```

AND THEN...

Concurrency

Timing assertions

Other assertions on performance

Quality of service assertions

DESIGN BY CONTRACT

Confluence of work from:

- **Axiomatic semantics of programming (Hoare 1969-1972)**
- **“Proof of correctness of Data Representations” (Hoare 1972)**
- **“Constructive approach” (Dijkstra 1976)**
- **Abrial’s Z (197)**
- **Abstract data types**
- **Object-oriented programming**
- **Reuse**

AN EXPLOSIVE COCKTAIL

Classes

Contracts

Dynamic aliasing

Procedures (state-changing operations)

Inheritance

Polymorphism and dynamic binding