

# Eiffel: a language for software engineering

Bertrand Meyer

LASER 2012



## The software of the future



### Product quality

- Correctness
- Robustness
- Security
- Efficiency

### Process quality

- Fast development
- No semantic gap ("impedance mismatch") between developers and other stakeholders
- Self-validating, self-testing
- Ease of change
- Reusability

## Where is Eiffel used?



Finance

Aerospace

Networking systems

Health care

Enterprise systems

Education (including introductory programming)

Often: lots of other solutions tried before!

3

## Eiffel: Method, Language, Environment



### *Method:*

- Applicable throughout the lifecycle
- Object-oriented
- Seamless development
- Based on Design by Contract™ principles

### *Language:*

- Full power of object technology
- Simple yet powerful, numerous original features
- ISO standard (2006)
- Supports full concurrency

### *Environment* (EiffelStudio):

- Integrated, provides single solution, including analysis and modeling
- Lots of platforms (Unix, Windows, VMS, .NET...)
- Open and interoperable

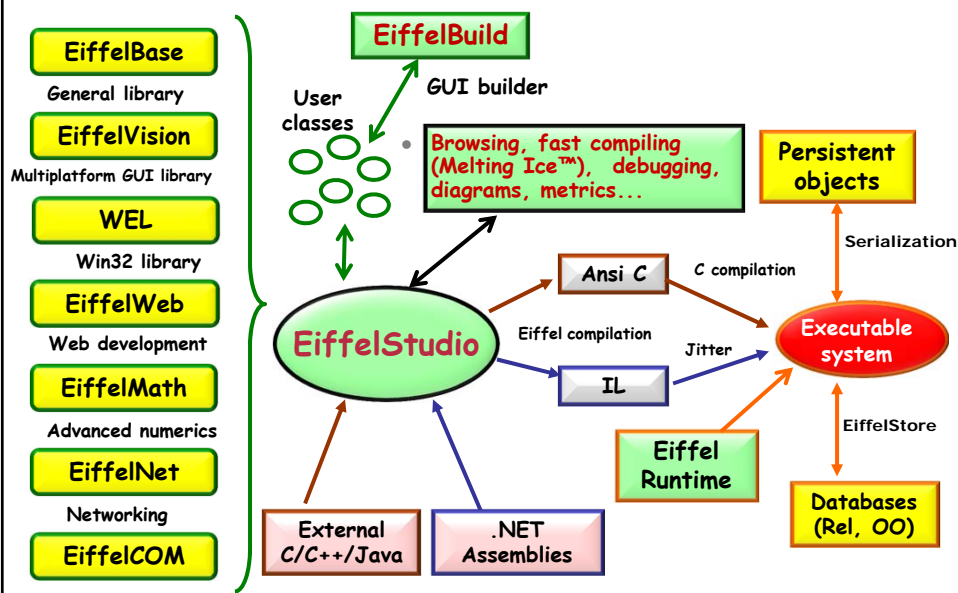
4

## The Eiffel method: some principles

- Abstract data types
  - Information hiding
  - Seamlessness, Reversibility
  - Design for reuse
  - Design by Contract
  - Concurrency as natural extension of sequential programming
  - Open-Closed principle
  - Single Choice principle
  - Single Model/Single Product principle
  - Uniform Access principle
  - Command-Query Separation principle
  - Option-Operand Separation principle
  - Style matters
- ... See next...

5

## EiffelStudio



6

## Eiffel is not...

Model-driven development

Functional programming

DSLs

Use-case-driven design

7

## Designing from use cases



8

## The competition



Rational Rose

SAP

SPARK

9

## Language versions



Eiffel 1, 1986

Classes, contracts, genericity, single and multiple inheritance, garbage collection, ...

Eiffel 2, 1988 (*Object-Oriented Software Construction*)

Exceptions, constrained genericity

Eiffel 3, 1990-1992 (*Eiffel: The Language*)

Basic types as classes, infix & prefix operators...

Eiffel 4, 1997

"Precursor" and agents

Eiffel 5, ECMA Standard, 2005, revised 2006, and ISO standard, November 2006

[www.ecma-international.org/publications/standards/Ecma-367.htm](http://www.ecma-international.org/publications/standards/Ecma-367.htm)

Attached types, conversion, assigner commands...

10

## The Eiffel language



- **Classes**
- **Statically typed**
- **Uniform** type system, covering basic types
- **Agents**: objects encapsulating behavior
- Built-in **Design by Contract** mechanisms, incl. exceptions
- Simple and safe **concurrency**: SCOOP
- **Genericity**
- Inheritance, single and **multiple**
- **Void safety**
- **Conversion**
- **Covariance**
- **"Once"** mechanisms, replacing statics and globals

11

## Learning Eiffel



- Simple syntax, no cryptic symbols  
Eiffel programmers know all of Eiffel
- Wide variety of user backgrounds  
*"If you can write a conditional,  
you can write a contract"*
- Fast learning curve
- Lots of good models to learn from
- Strong style rules
- May need to "unlearn" needless tricks
- Borrows less from C than you'd think

12

## Teaching

First Java program:

```
class First {  
    public static void main(String args[])  
    { System.out.println("Hello World!"); } }
```

You'll understand  
when you grow up!

Do as I say,  
not as I do

13

## What is not in Eiffel

- Goto
- Functions as arguments (but: agents)
- Pointer arithmetic
- Special increment syntax, e.g.  $x++$ ,  $++x$
- In-class feature overloading
- Direct access to object fields:  $x.a := v$
- Mechanisms that directly conflict with O-O principles, e.g. static functions

14

## Dogmatism and flexibility



### Dogmatic where it counts:

- Information hiding (e.g. no  $x.a := v$ )
- Overloading
- "One good way to do anything"
- Style rules

### Flexible when it makes no point to harass programmers:

- Give standard notations an O-O interpretation  
Examples:
  - $a + b$
  - $x.a := v$
- Syntax, e.g. semicolon

15

## Syntax conventions



Semicolon used as a separator (not terminator)

It's optional almost all the time. Just forget about it!

Style rules are an important part of Eiffel:

- Every feature should have a header comment
- Every class should have an **indexing** clause
- Layout, indentation
- Choice of names for classes and features

16



## More language design principles



Keywords are full English-language words, e.g. **require**  
(there is one exception: **elseif**)

Generally simplest version of work (**require**, not **requires**)

Strong style rules, e.g. indentation, choice of names, letter case (language itself is case-insensitive), comments...

Not minimalistic but **"One good way to do anything"**

Language evolution: it's OK to remove features

17

## Style of Eiffel language description



Specification on three levels:

- **Syntax**
- **Validity**
- **Semantics**

18

## Syntax: structure of texts



Syntactically illegal examples:

`x.a = b`

19

## Syntax description



BNF-like

Three kinds of production: aggregate, choice, list

Each non-terminal construct defined by exactly one production

No mixing!

20

## Syntax specification



```
Indexing  ≡ indexing Index_list
Index_list ≡ (Index_clause "; ..."*)
Index_clause ≡ Index Index_terms
Index ≡ Identifier "."
Index_terms ≡ {Index_value "; ..." +
Index_value ≡ Identifier | Manifest_constant
```

21

## Validity: constraints on syntactically legal texts



**Invalid example:**

`your_integer + your_boolean`

22

## Semantics: effect of valid texts, if defined



Incorrect example:

`x := Void`

`x.your_feature`

23

## Validity rules: if and only if



### Local Entity rule

*CRLE*

Let *ld* be the Local\_declarations part of a routine *r* in a class *C*.  
Let *locals* be the concatenation of every Identifier\_list of every  
Entity\_declaration\_group in *ld*. Then *ld* is valid if and only if  
every Identifier *e* in *ld* satisfies the following two conditions:

- 1 • No feature of *C* has *e* as its final name.
- 2 • No formal argument of *r* has *e* as its Identifier.

24

## Openness



Eiffel can be used as "component combinator" to package elements from different sources:

- Mechanisms for integrating elements in C, C++, Java, CIL (.NET)
- Interfaces and libraries: SQL, XML, UML (XMI), CORBA, COM, others
- Particularly extensive C/C++ interfacing
- Outside of .NET, compiles down to ANSI C code, facilitates support for C and C++ easier.
- On .NET, seamless integration with C#, VB .NET etc.

25

## The Eiffel language: there *is* a hidden agenda



*That you forget it even exists*

26



---

--

# The Eiffel method

27



## The Eiffel method: some principles

---

- Abstract data types
  - Information hiding
  - Seamlessness, Reversibility
  - Design for reuse
  - Design by Contract
  - Concurrency as natural extension of sequential programming
  - Open-Closed principle
  - Single Choice principle
  - Single Model/Single Product principle
  - Uniform Access principle
  - Command-Query Separation principle
  - Option-Operand Separation principle
  - Style matters
- ... See next...

28

## Traditional lifecycle model

### Rigid model:

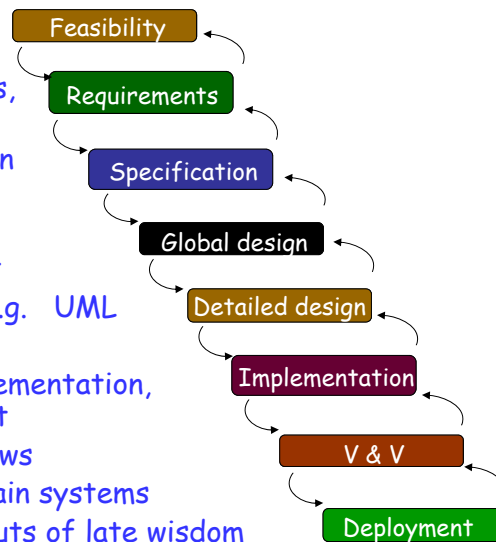
- Waterfall: separate tasks, impedance mismatches
- Variants, e.g. spiral, retain some of the problems

### 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

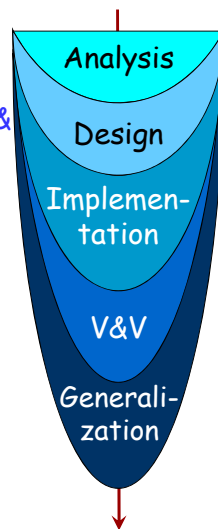


29

## The Eiffel model

### 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 & forth
- Saves money: invest in single set of tools
- Boosts quality



Example classes:

*PLANE, ACCOUNT, TRANSACTION...*

*STATE, COMMAND...*

*HASH\_TABLE...*

*TEST\_DRIVER...*

*TABLE...*

30

## Seamlessness



### Seamlessness Principle

Software development should rely on a single set of notations & tools

31

## Reversibility



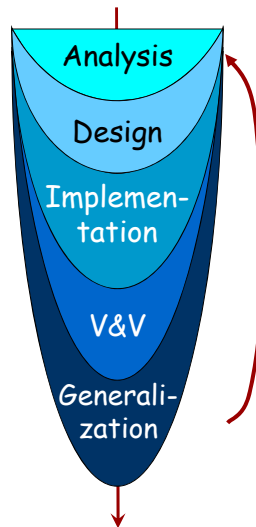
### Reversibility Principle

The software development process, notations and tools should allow making changes at any step in the process

32



## The seamless, reversible model



Example classes:

*PLANE, ACCOUNT,  
TRANSACTION...*

*STATE, COMMAND...*

*HASH\_TABLE...*

*TEST\_DRIVER...*

*TABLE...*

33

## Analysis classes

deferred class *VAT* inherit

*TANK*

feature

*in\_valve, out\_valve: VALVE*

*fill*

**require** -- Fill the vat.  
*in\_valve.open*  
*out\_valve.closed*

**deferred**

**ensure**  
*in\_valve.closed*  
*out\_valve.closed*  
*is\_full*

**end**

*empty, is\_full, is\_empty, gauge, maximum,*

**invariant**

*is\_full = (gauge >= 0.97 \* maximum) and (gauge <= 1.03 \* maximum)*

**end**

Precondition

Specified, not  
implemented

Postcondition

Class invariant

34

## Single model

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

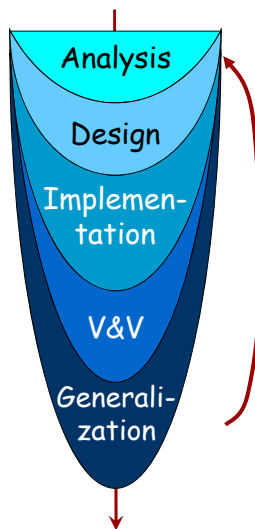
Use **tools** to extract the appropriate **views**.

### Single Model Principle

**All the information  
about a software system  
should be in the software text**

35

## The seamless, reversible model

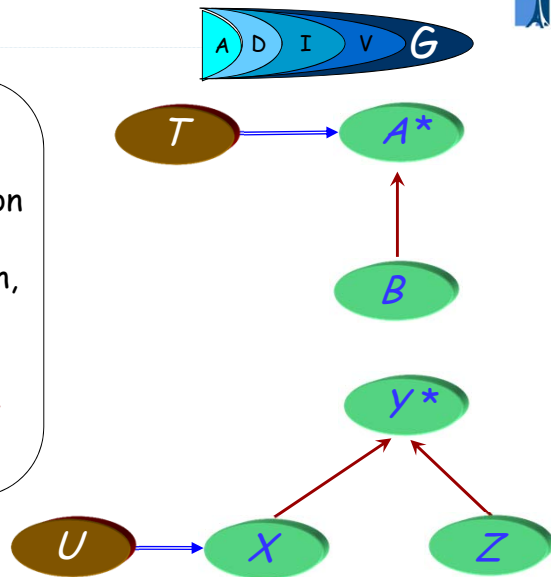


36

## Generalization

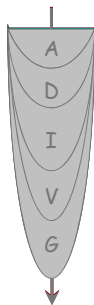
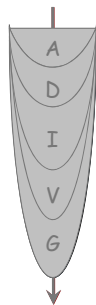
Prepare for reuse:

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

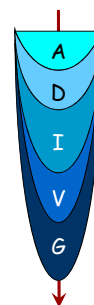
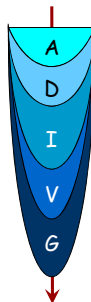


37

## The cluster model



Mix of sequential and concurrent engineering



Permits dynamic reconfiguration

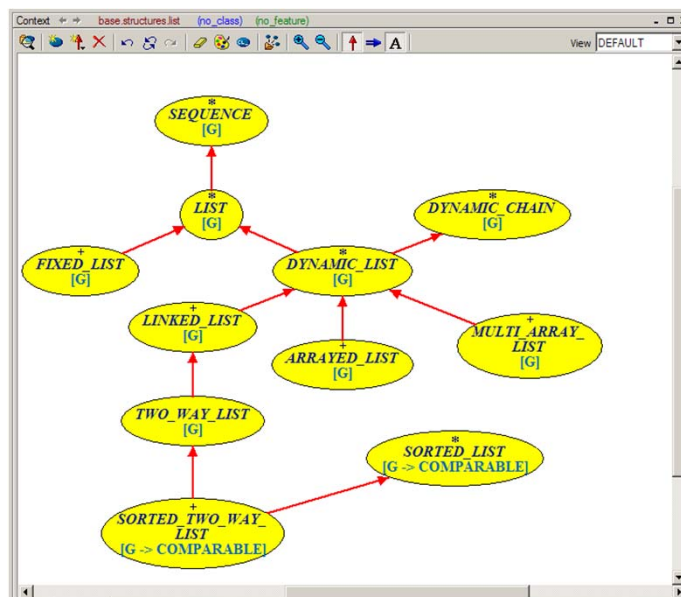
38

## Tool support for seamless development

- Diagram Tool
  - System diagrams can be produced automatically from software text
  - Works both ways: update diagrams or update text - other view immediately updated
- No need for separate UML tool
- Metrics Tool
- Profiler Tool
- Documentation generation tool
- ...

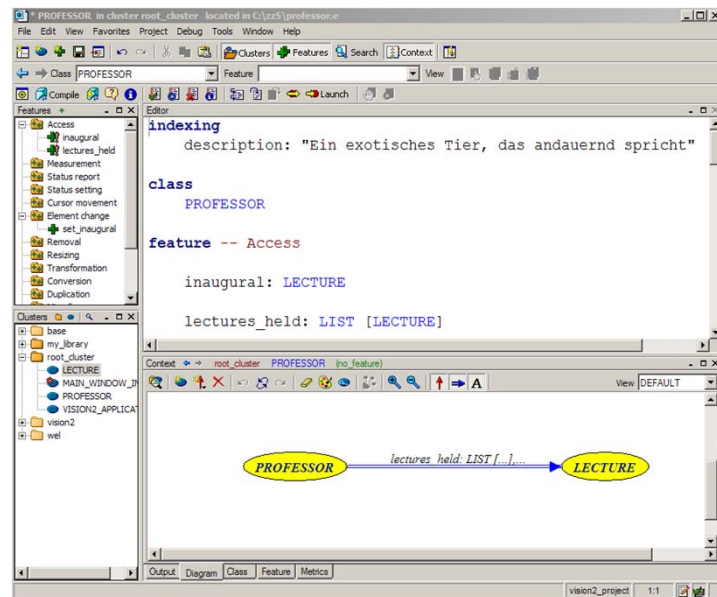
39

## EiffelStudio diagram tool



40

## Text-graphics equivalence



41

## Equivalence

### Equivalence Principle

Textual, graphical and other views  
should all represent the same model

42

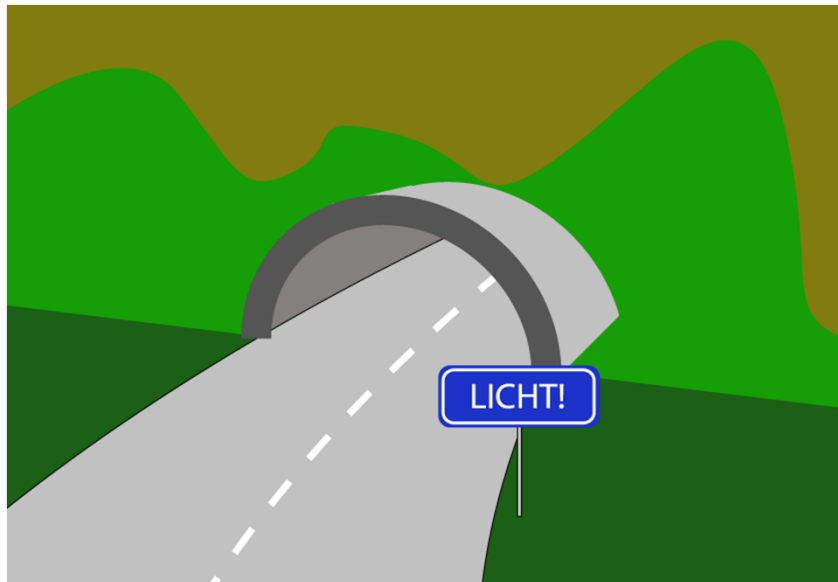
## Command-Query separation principle



Asking a question  
should not change the answer

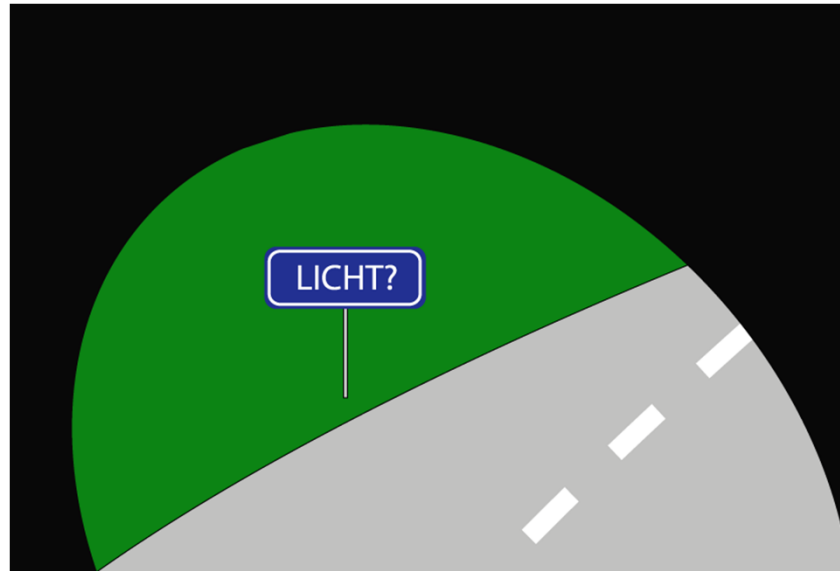
43

## A command



44

## A query



45

## Command-Query separation principle

Asking a question  
should not change the answer

46

## Command-Query separation



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.

47

## Command-Query Separation



Asking a question  
should not change the answer!

48



## 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.

49

## Command-query separation



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

$io.read\_integer$

$n := io.last\_integer$

50

## The class



From the module viewpoint:

- Set of available services ("features")
- Information hiding
- Classes may be **clients** of each other
- A class may extend another, through inheritance

From the type viewpoint:

- Describes a set of run-time objects (instances of the class)
- Used to declare variables (more generally, *entities*)  
 $x: C$
- Static type checking
- A class may specialize another, through inheritance

51

## Language style



Compatibility principle

Traditional notations should be supported  
with an O-O semantics

52

## Infix and prefix operators



In

$a - b$

the  $-$  operator is "infix"  
(written between operands)

In

$- b$

the  $-$  operator is "prefix"  
(written before the operand)

53

## The object-oriented form of call



*some\_target.some\_feature(some\_arguments)*

For example:

*my\_figure.display*

*my\_figure.move(3, 5)*

*x := a.plus(b)*    *???????*

54

## Operator features

expanded class *INTEGER* feature

```
plus alias "+" (other: INTEGER): INTEGER
    -- Sum with other
do ... end

times alias "*" (other: INTEGER): INTEGER
    -- Product by other
do ... end

minus alias "-" : INTEGER
    -- Unary minus
do ... end

...
end
```

Calls such as *i.plus(j)* can now be written *i + j*

55

## Assignment commands

It is possible to define a query as

```
temperature: REAL assign set_temperature
```

Then the syntax

```
x.temperature := 21.5
```

Not an assignment, but a  
procedure call

is accepted as an abbreviation for

```
x.set_temperature(21.5)
```

Retains **contracts** and any other supplementary operations

56

## Using the bracket alias

In class *ARRAY*[*G*]:

```
item alias "[" (i: INTEGER): G assign put  
  require  
    i >= lower and i <= count  
  do ... end  
  
put (x: G; i: INTEGER): G  
  require  
    i >= lower and i <= count  
  do ... end  
    a . put (a . item (i) + 1, i)  
    a . item (i) := a . item (i) + 1  
    a [i] := a [i] + 1
```

Not an assignment!

57

## Bracket alias

```
population ["Procchio"] := 366
```

```
table [a, b, c] := d
```

58

## Array access

Object-oriented forms:

```
a: ARRAY[T]  
a•put(x, 23)  
x := a•item(23)
```

Above mechanisms make the following synonyms possible:

```
a[23] := x
```

```
x := a[23]
```

Usual form:

```
a[i] := a[i] + 1
```

Object-oriented form:

```
a•put(a•item(i) + 1, i)
```

59

Design by Contract

60

## Design by Contract



### Contract Principle

Every software element  
should be characterized  
by a precise specification

61

## Andrew Binstock, Dr. Dobb's, 26 Aug 2012



<http://bit.ly/O480Ob>

(slightly abridged)

I've found myself constantly frustrated by the feeling that no matter how much I test my code, I can't be sure that it's right. The best I can say is that it is probably right. But when I write code for others, I want it to be completely reliable. This concern has led me to embrace tools that enforce correctness. Long ago, I adopted Bertrand Meyer's concept of design-by-contract (DBC), which suggests that every function test for preconditions, postconditions, and invariants. In Java, I do this with Guava. My methods tend to have tests, especially at the beginning to check each parameter carefully. I test invariants and post-conditions primarily in unit tests, which is probably not ideal, but moves some of the validation clutter out of the code.

62

## Design by Contract: applications



- Getting the software right
- Analysis
- Design
- Implementation
- Debugging
- Testing
- Management
- Maintenance
- Documentation

63

## Design by Contract: the basic idea



Every software element is intended to satisfy a certain goal, for the benefit of other software elements (and ultimately of human users)

This goal is the element's **contract**

The contract of any software element should be

- Explicit
- Part of the software element itself

64



## A counter-example: Ariane 5, 1996



(See: Jean-Marc Jézéquel and Bertrand Meyer: *Design by Contract: The Lessons of Ariane*, IEEE Computer, January 1997, also at <http://www.eiffel.com>)

37 seconds into flight, exception in Ada program not processed; order given to abort the mission. Ultimate cost in billions of euros

Cause: incorrect conversion of 64-bit real value ("horizontal bias" of the flight) into 16-bit integer

Systematic analysis had "proved" that the exception could not occur!

65

## Ariane-5 (continued)



It was a REUSE error:

- The analysis was correct - for Ariane 4 !
- The assumption was documented - in a design document !

With assertions, the error would almost certainly be detected by either static inspection or testing:

```
integer_bias(b: REAL): INTEGER
  require
    representable(b)
  do
    ...
  ensure
    equivalent(b, Result)
end
```

66

## The contract view of software construction



Constructing systems as structured collections of cooperating software elements — suppliers and clients — cooperating on the basis of clear definitions of obligations and benefits

These definitions are the contracts

67

## Contracts for analysis



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

68

## Contracts for analysis

```

deferred class VAT inherit
  TANK
  feature
    in_valve, out_valve: VALVE
    fill
      -- Fill the vat.
      require
        in_valve.open
        out_valve.closed
      deferred
        ensure
          in_valve.closed
          out_valve.closed
          is_full
        end
    empty, is_full, is_empty, gauge, maximum,
  invariant
    is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)
end

```

**Precondition**

**Specified, not implemented**

**Postcondition**

**Class invariant**

69

## A class without contracts

```

class
  ACCOUNT
  feature -- Access
    balance: INTEGER
    -- Balance

    Minimum_balance: INTEGER = 1000
    -- Minimum balance
  feature {NONE} -- Deposit and withdrawal
    add (sum: INTEGER)
      -- Add sum to the balance.
      do
        balance := balance + sum
      end

```

**Secret features**

70

## A class without contracts

```
feature -- Deposit and withdrawal operations
  deposit(sum: INTEGER)
    -- Deposit sum into the account.
    do
      add(sum)
    end

  withdraw(sum: INTEGER)
    -- Withdraw sum from the account.
    do
      add(- sum)
    end

  may_withdraw(sum: INTEGER): BOOLEAN
    -- Is it permitted to withdraw sum from the account?
    do
      Result := (balance - sum >= Minimum_balance)
    end
end
```

71

## Introducing contracts

```
class
  ACCOUNT
create
  make
feature {NONE} -- Initialization
  make(initial_amount: INTEGER)
    -- Set up account with initial_amount.

    require
      large_enough: initial_amount >= Minimum_balance
    do
      balance := initial_amount

      ensure
        balance_set: balance = initial_amount
      end
end
```

72

## Introducing contracts

**feature** -- Access

*balance*: *INTEGER*  
-- Balance

*Minimum\_balance*: *INTEGER* = 1000  
-- Lowest permitted balance

**feature** {*NONE*} -- Implementation of deposit and withdrawal

*add*(*sum*: *INTEGER*)

-- Add *sum* to the *balance*.

**do**

*balance* := *balance* + *sum*

**ensure**

increased: *balance* = old *balance* + *sum*

**end**

73

## Introducing contracts

**feature** -- Deposit and withdrawal operations

*deposit*(*sum*: *INTEGER*)

-- Deposit *sum* into the account.

**require**

not\_too\_small: *sum* >= 0

**do**

*add*(*sum*)

**ensure**

increased: *balance* = old *balance* + *sum*

**end**

Precondition

Postcondition

74

## Introducing contracts

```

withdraw(sum: INTEGER)
  -- Withdraw sum from the account.
  require
    not_too_small: sum >= 0
    not_too_big: sum <= balance - Minimum_balance
  do
    add(-sum)
    -- i.e. balance := balance - sum
  ensure
    decreased: balance = old balance - sum
  end

```

Value of *balance*, captured on entry to routine

75

## The contract

<i>withdraw</i>	OBLIGATIONS	BENEFITS
<i>Client</i>	(Satisfy precondition:) Make sure <i>sum</i> is neither too small nor too big	(From postcondition:) Get account updated with <i>sum</i> withdrawn
<i>Supplier</i>	(Satisfy postcondition:) Update account for withdrawal of <i>sum</i>	(From precondition:) Simpler processing: may assume <i>sum</i> is within allowable bounds

76

## The imperative and the applicative

<b>do</b> <i>balance := balance - sum</i>	<b>ensure</b> <i>balance = old balance - sum</i>
<b>PRESCRIPTIVE</b>	<b>DESCRIPTIVE</b>
How?	What?
Operational	Denotational
<b>Implementation</b>	<b>Specification</b>
Command	Query
Instruction	Expression
Imperative	Applicative

77

## Introducing contracts

```

may_withdraw(sum: INTEGER): BOOLEAN
  -- Is it permitted to withdraw sum from account?
do
  Result := (balance - sum >= Minimum_balance)
end

```

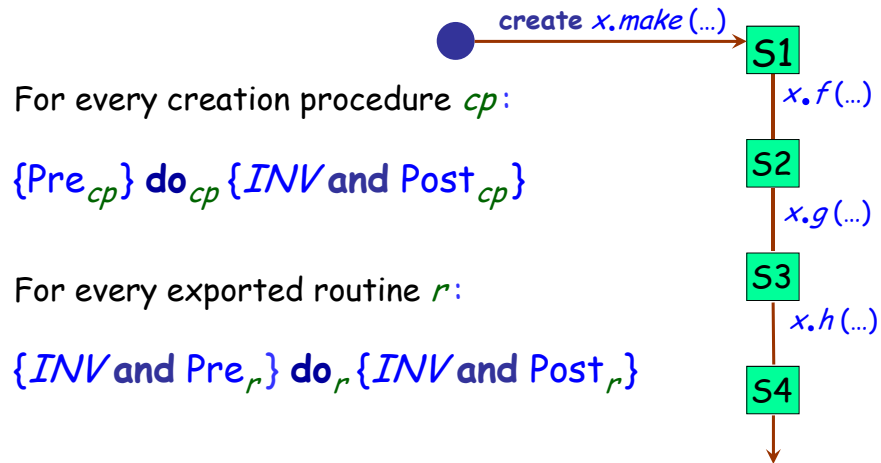
```

invariant
  not_under_minimum: balance >= Minimum_balance
end

```

78

## The correctness of a class



79

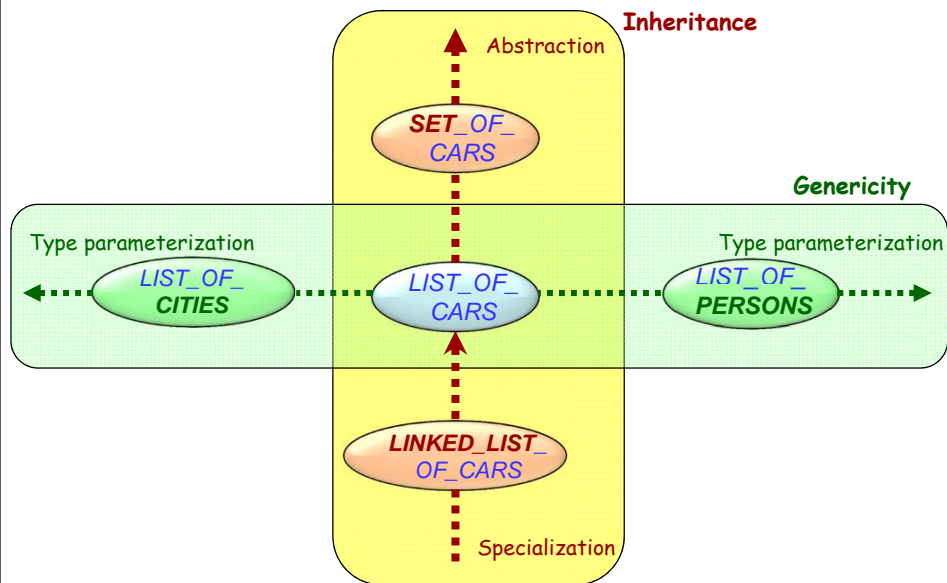
## Genericity & inheritance

"Genericity versus Inheritance", OOPSLA 1986

80



## Extending the basic notion of class



81

## Genericity: Ensuring type safety

How can we define consistent "container" data structures, e.g. list of accounts, list of points?

Dubious use of a container data structure:

What if wrong?

*c* : *CITY* ; *p* : *PERSON*

*cities* : *LIST*...

*people* : *LIST*...

-----  
*people.extend* (*p*)

*cities.extend* (*c*)

*c* := *cities.last*

*c.some\_city\_operation*

82

## A generic class

```
class LIST[G] feature
  extend(x: G) ...
  last: G ...
end
```

Formal generic parameter

To use the class: obtain a generic derivation, e.g.

```
cities: LIST[CITY]
```

Actual generic parameter

83

## Using generic derivations

```
cities: LIST[CITY]  
people: LIST[PERSON]  
c: CITY  
p: PERSON  
...
```

```
cities.extend(c)  
people.extend(p)
```

```
c := cities.last  
c.some_city_operation
```

### STATIC TYPING

The compiler will reject:

- *people*.extend(*c*)
- *cities*.extend(*p*)

84

## Static typing



### Type-safe call (during execution):

A feature call  $x.f$  such that the object attached to  $x$  has a feature corresponding to  $f$ .

[Generalizes to calls with arguments,  $x.f(a, b)$ ]

### Static type checker:

A program-processing tool (such as a compiler) that guarantees, for any program it accepts, that any call in any execution will be *type-safe*.

### Statically typed language:

A programming language for which it is possible to write a *static type checker*.

85

## Using genericity



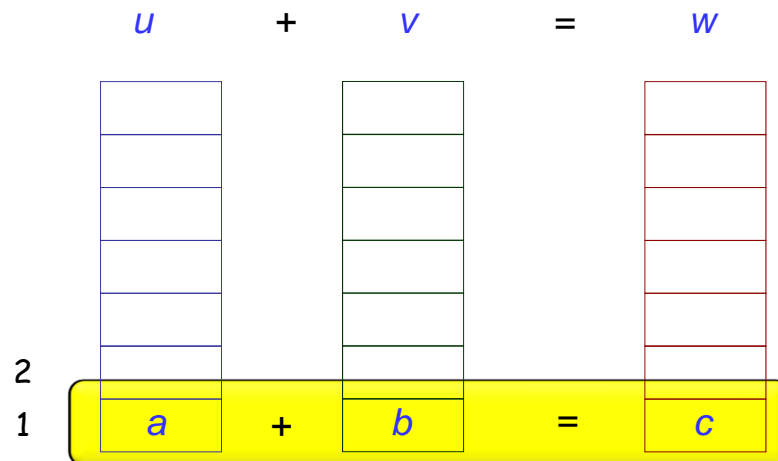
*LIST* [*CITY*]  
*LIST* [*LIST* [*CITY*]]  
...

A type is no longer exactly the same thing as a class!

(But every type remains **based** on a class.)

86

## Adding two vectors



87

## Genericity + inheritance 2: Constrained genericity

```

class VECTOR [G          ] feature
  plus alias "+" (other: VECTOR [G]): VECTOR [G]
    -- Sum of current vector and other
  require
    lower = other.lower
    upper = other.upper
  local
    a, b, c: G
  do
    ... See next ...
  end
  ... Other features ...
end

```

88

## Constrained genericity


Body of *plus* alias "+":

```
create Result.make(lower, upper)
from
  i := lower
until
  i > upper
loop
  a := item(i)
  b := other.item(i)
  c := a + b -- Requires "+" operation on G!
  Result.put(c, i)
  i := i + 1
end
```

89

## The solution

Declare class *VECTOR* as

```
class VECTOR[G  NUMERIC] feature
  ... The rest as before ...
end
```

Class *NUMERIC* (from the Kernel Library) provides features *plus* alias "+", *minus* alias "-" and so on.

90

## Improving the solution

Make *VECTOR* itself a descendant of *NUMERIC*,  
effecting the corresponding features:

```
class VECTOR[G -> NUMERIC] inherit
  NUMERIC
feature
  ... Rest as before, including infix "+"...
end
```

Then it is possible to define

```
v: VECTOR[INTEGER]
vv: VECTOR[VECTOR[INTEGER]]
vvv: VECTOR[VECTOR[VECTOR[INTEGER]]]
```

91

## The class invariant

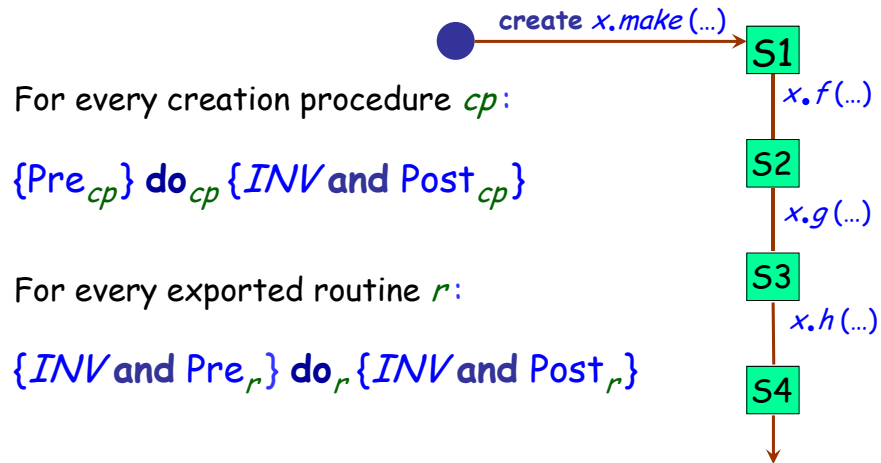
Consistency constraint applicable to all instances of a class.

Must be satisfied:

- After creation
  - After execution of any feature by any client
- Qualified calls only: *x.f(...)*

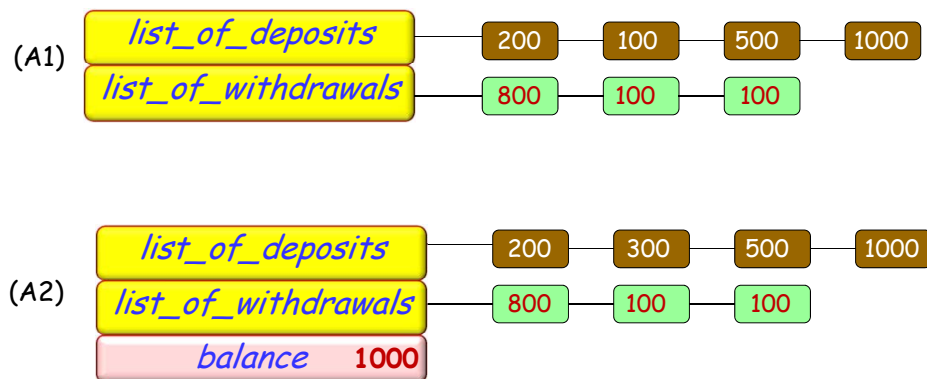
92

## The correctness of a class



93

## Uniform Access



$balance = deposits.total - withdrawals.total$

$a: ACCOUNT$

....

$print(a.balance)$

94

## What are contracts good for?



Writing correct software (analysis, design,  
implementation, maintenance, reengineering)  
Documentation (the "contract" form of a class)  
Effective reuse  
Controlling inheritance  
Preserving the work of the best developers  
Proofs

---

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

95

## A contract violation is not a special case



For special cases  
(e.g. "if the sum is negative, report an error...")

use standard control structures, such as **if ... then ... else...**

A run-time assertion violation is something else: the  
manifestation of

A DEFECT ("BUG")

96



## Contracts and quality assurance



Precondition violation: Bug in the client.

Postcondition violation: Bug in the supplier.

Invariant violation: Bug in the supplier.

$$\{P\} \quad A \quad \{Q\}$$

97

## Contracts: run-time effect



Compilation options (per class, in Eiffel):

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

98

## Contracts for testing and debugging

Contracts express implicit assumptions behind code

- A bug is a discrepancy between intent and code
- Contracts state the intent!

In EiffelStudio: select compilation option for run-time contract monitoring at level of:

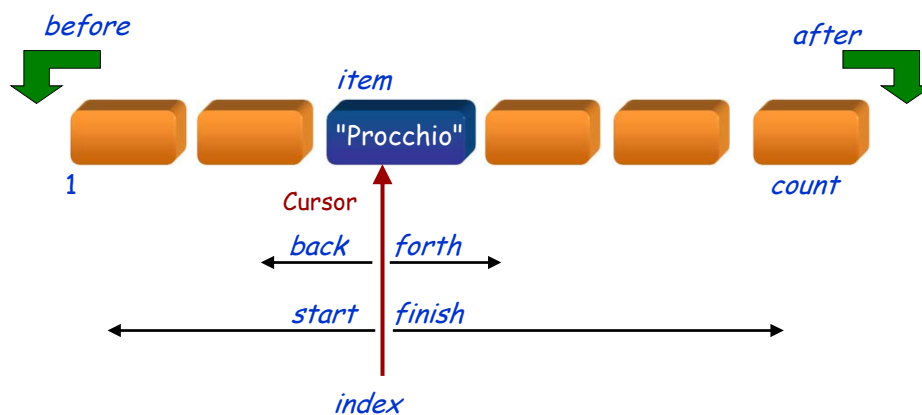
- Class
- Cluster
- System

May disable monitoring when releasing software

A revolutionary form of quality assurance

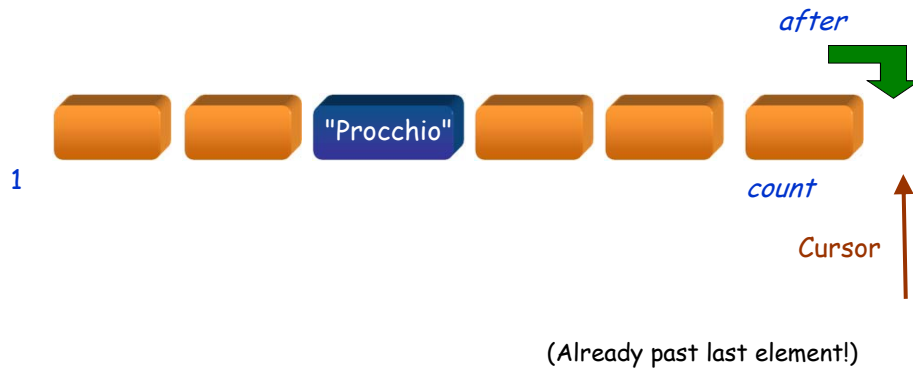
99

## Lists in EiffelBase



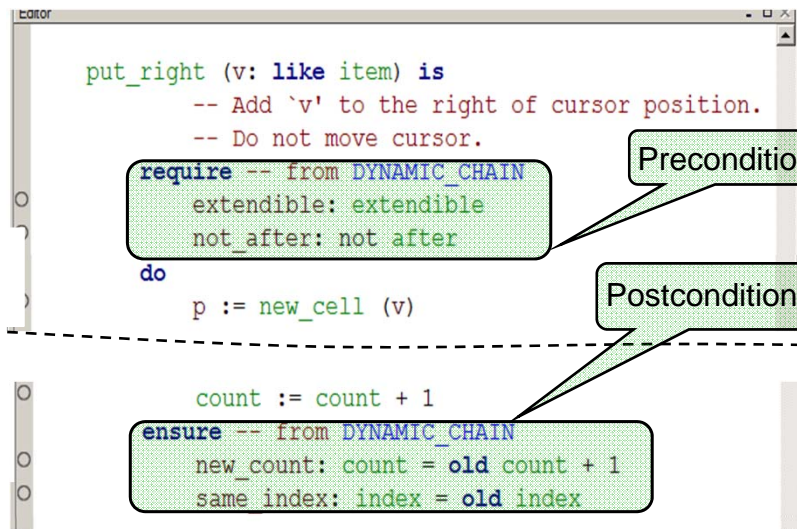
100

## Trying to insert too far right



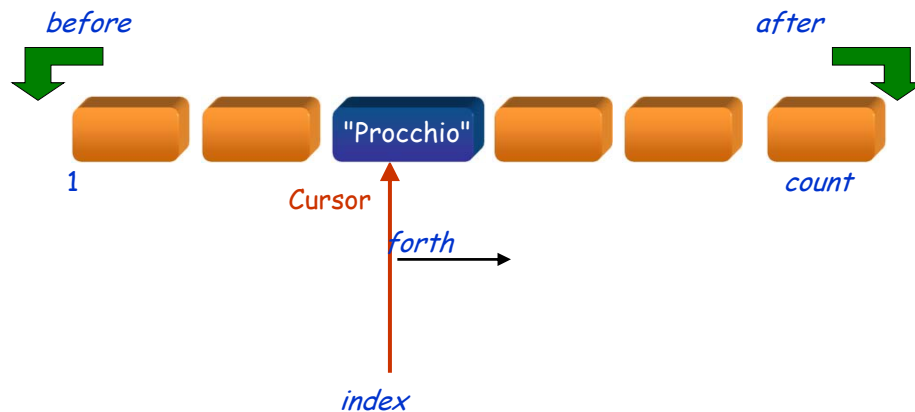
101

## A command and its contract



102

## Moving the cursor forward



103

## Two queries, and command *forth*

```
Editor
feature -- Status report

  after: BOOLEAN
    -- Is there no valid cursor position to the right of cursor?

  before: BOOLEAN
    -- Is there no valid cursor position to the left of cursor?

feature -- Cursor movement

  forth
    -- Move to next position; if no next position,
    -- ensure that 'exhausted' will be true.
    require -- from LINEAR
      not_after: not after
    ensure then
      moved_forth: index = old index + 1
```

104

## Where the cursor may go



Valid cursor positions

105

## From the invariant of class *LIST*

```
Editor
invariant

prunable: prunable
before_definition: before = (index = 0)
after_definition: after = (index = count + 1)
-- from CHAIN

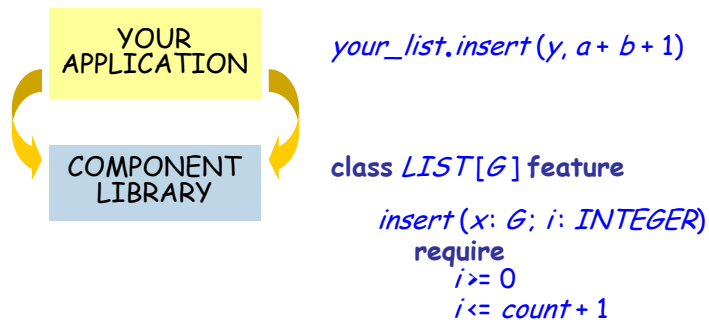
non_negative_index: index >= 0
index_small_enough: index <= count + 1
```

Valid cursor positions

106

## Contracts and bug types

Preconditions are particularly useful to find bugs in **client** code:



107

## Contracts and quality assurance

Use run-time assertion monitoring for quality assurance, testing, debugging.

Compilation options (reminder):

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

108

## Contracts and quality assurance

Contracts enable QA activities to be based on a precise description of what they expect.

Profoundly transform the activities of testing, debugging and maintenance.

*"I believe that the use of Eiffel-like module contracts is the most important non-practice in software world today. By that I mean there is no other candidate practice presently being urged upon us that has greater capacity to improve the quality of software produced. ... This sort of contract mechanism is the sine-qua-non of sensible software reuse. "*

Tom de Marco, IEEE Computer, 1997

109

## Automatic testing

AutoTest (part of EiffelStudio):

- Test generation      Test cases produced automatically from software
- Test extraction      Test cases produced automatically from failures
- Manual testing      Test cases produced explicitly by developers or testers

110

## AutoTest: Test generation

- Input: set of classes + testing time
- Generates instances, calls routines with automatically selected args
- Oracles are contracts:
  - Direct precondition violation: skip
  - Postcondition/invariant violation: bingo!
- Value selection: Random+ (use special values such as 0, +/-1, max and min)
- Add manual tests if desired
- Any test (manual or automated) that fails becomes part of the test suite

Ilinca Ciupa  
Andreas Leitner  
Manuel Oriol  
Yi Wei  
Arno Fiva  
et al.

111

## Contracts and documentation

**Contract view:** Simplified form of class text, retaining interface elements only:

- Remove any non-exported (private) feature

For the exported (public) features:

- Remove body (do clause)
- Keep header comment if present
- Keep contracts: preconditions, postconditions, invariant
- Remove any contract clause that refers to a secret feature

(This raises a problem; can you see it?)

112



## The next step



Proofs

113

## Flat, interface



**Flat view of a class:** reconstructed class with all the features at the same level (immediate and inherited). Takes renaming, redefinition etc. into account.

**The flat view is an inheritance-free client-equivalent form of the class.**

**Interface view:** the contract view of the flat view. Full interface documentation.

114

## Uses of the contract & interface forms



Documentation, manuals

Design

Communication between developers

Communication between developers and managers

115

## Contracts and inheritance



Issues: what happens, under inheritance, to

- Class invariants?
- Routine preconditions and postconditions?

116

## Invariants

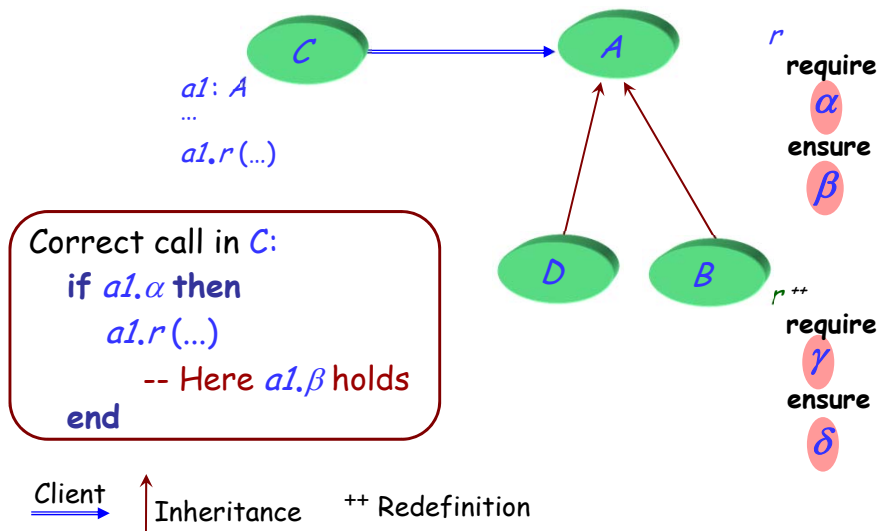
### Invariant Inheritance rule:

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

Accumulated result visible in flat and interface forms.

117

## Contracts and inheritance



118

## Assertion redeclaration rule



When redeclaring a routine, we may only:

- Keep or weaken the precondition
- Keep or strengthen the postcondition

119

## Assertion redeclaration rule in Eiffel



A simple language rule does the trick!

Redefined version 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*

120

## Exception handling



Two concepts:

- **Failure**: a routine, or other operation, is unable to fulfill its contract.
- **Exception**: an undesirable event occurs during the execution of a routine — as a result of the **failure** of some operation called by the routine.

121

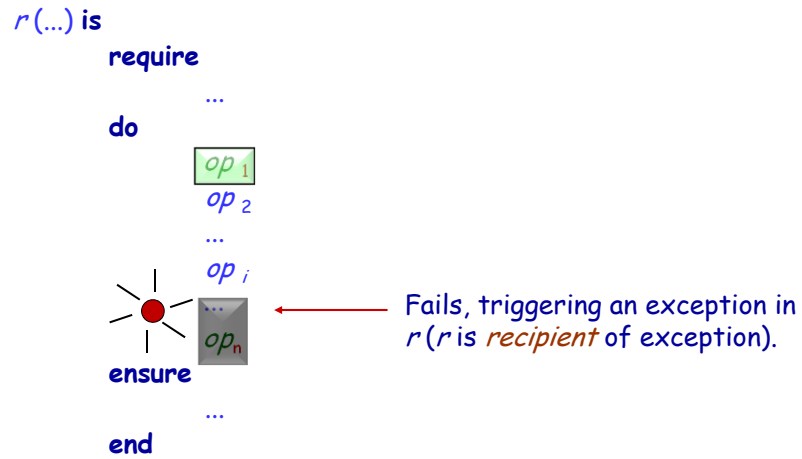
## The original strategy



```
r(...) is
  require
    ...
  do
    op1
    op2
    ...
    opi
    ...
    opn
  ensure
    ...
end
```

122

## Not going according to plan



123

## Handling exceptions

Safe exception handling principle:

There are only two acceptable ways to react for the recipient of an exception:

- Concede failure, and trigger an exception in caller:  
"Organized Panic"
- Try again, using a different strategy (or repeating the same strategy):  
"Retrying"

(Rare third case: false alarm)

124

## Exception mechanism



Two constructs:

- A routine may contain a **rescue** clause.
- A rescue clause may contain a **retry** instruction.

A **rescue** clause that does not execute a **retry** leads to failure of the routine (this is the organized panic case).

125

## Transmitting over an unreliable line (1)



```
Max_attempts: INTEGER = 100

attempt_transmission(message: STRING)
  -- Transmit message in at most
  -- Max_attempts attempts.
  local
    failures: INTEGER
  do
    unsafe_transmit(message)
  rescue
    failures := failures + 1
    if failures < Max_attempts then
      retry
    end
  end
end
```

126

## Transmitting over an unreliable line (2)

```
Max_attempts: INTEGER = 100
failed: BOOLEAN

attempt_transmission(message: STRING)
    -- Try to transmit message;
    -- if impossible in at most Max_attempts
    -- attempts, set failed to true.
    local
        failures: INTEGER
    do
        if failures < Max_attempts then
            unsafe_transmit(message)
        else
            failed := True
        end
    rescue
        failures := failures + 1
        retry
    end
```

127

## The assertion language

Assertions in Eiffel use boolean expressions of the programming language, plus **old** in postconditions

Consequences of this design decision:

- Assertions can be used for both
  - Static checking, in particular **proofs**
  - Dynamic evaluation, as part of **testing**
- No first- or higher-order predicate calculus
- Can use query calls (functions, attributes)
  - Must guarantee absence of **side effects!**

128



## Eiffel Model Library (MML)

Bernd Schoeller, Tobias Widmer, Nadia Polikarpova

Classes correspond to mathematical concepts:

*SET[G], FUNCTION[G, H], TOTAL\_FUNCTION[G, H],  
RELATION[G, H], SEQUENCE[G], ...*

Completely applicative: no attributes (fields), no  
implemented routines (all completely deferred)

Specified with contracts (unproven) reflecting  
mathematical properties

Expressed entirely in Eiffel

129

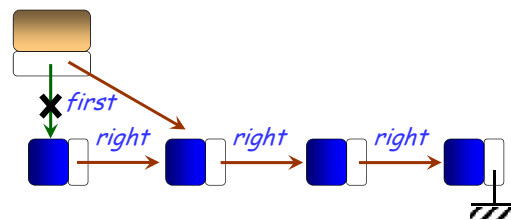
## Specifying lists

class

*LINKED\_LIST [G]*

feature

...  
*remove\_front*



-- Remove first item.

require

not *empty*

do

*first* := *first.right*

ensure

*model* = old *model.tail*

*count* = old *count* - 1

*first* = old *item* (2)

end

end

130

## Example MML class

```
class SEQUENCE[G] feature
  count: NATURAL
  last: G
  extended (x): SEQUENCE[G]
  ensure
    Result.count = count + 1
    Result.last = x
    Result.sub (1, count) ~ Current
  mirrored: SEQUENCE[G]
  ensure
    Result.count = count
  ...
end
```

131

## Principles

Very simple mathematics only

- Logic
- Set theory

132

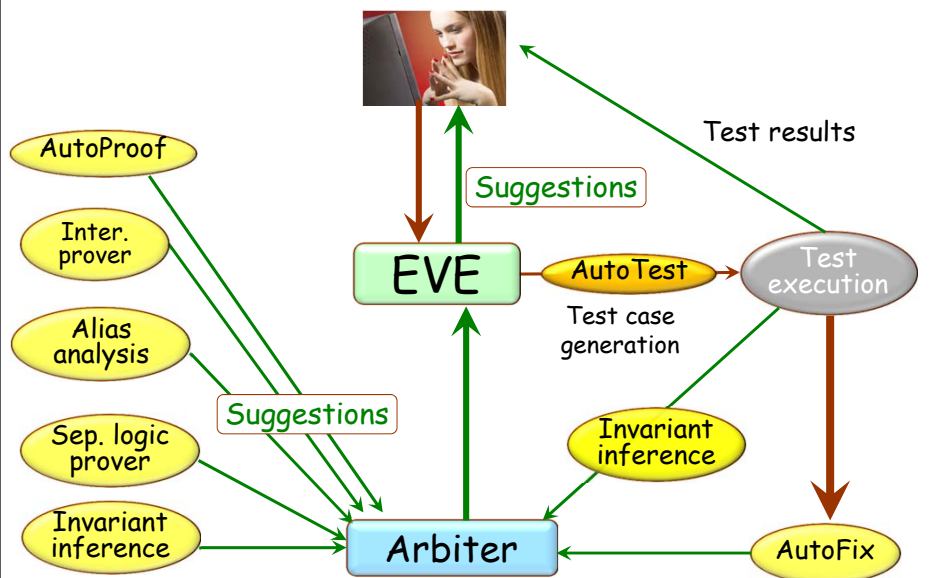
## EiffelBase+

Nadia Polikarpova

In progress: library of fully specified (MML) classes,  
covering fundamental data structures and algorithms, and  
designed for verification: tests and proofs

133

## Verification As a Matter Of Course



134

## Contracts as a management tool



High-level view of modules for the manager:

- Follow what's going on without reading the code
- Enforce strict rules of cooperation between units of the system
- Control outsourcing

135

## Managerial benefits



- Library users can trust documentation
- They benefit from preconditions to validate their own code
- Component-based development possible on a solid basis
- More accurate estimates of test effort
- Black-box specification for free
- Designers who leave bequeath not only code but intent
- Common vocabulary between stakeholders: developers, managers, customers...

136

## Concurrency in Eiffel: SCOOP



No data races

137

## Concurrency in Eiffel: SCOOP



No data races

138

## Concurrency in Eiffel: SCOOP



No data races

139

## Concurrency in Eiffel: SCOOP



No data races

140

## Concurrency in Eiffel: SCOOP



No data races

141

## Concurrency in Eiffel: SCOOP



No data races

142

## Concurrency in Eiffel: SCOOP



No data races

143

## Concurrency in Eiffel: SCOOP



No data races

144



## Concurrency in Eiffel: SCOOP



No data races

145

## Concurrency in Eiffel: SCOOP



No data races

146

## Concurrency in Eiffel: SCOOP



No data races

147

## Concurrency in Eiffel: SCOOP



No data races

148



No data races



No data races



No data races

151



No data races

152



No data races

153



No data races

154



# No data races

155



# No data races

156



## Avoid a void

Bertrand Meyer

With major contributions by **Emmanuel Stapf** &  
**Alexander Kogtenkov** (Eiffel Software)

and the ECMA TG4 (Eiffel) committee,  
plus gratefully acknowledged influence of Spec#,  
especially through Erik Meijer & Rustan Leino

### Basic O-O operation

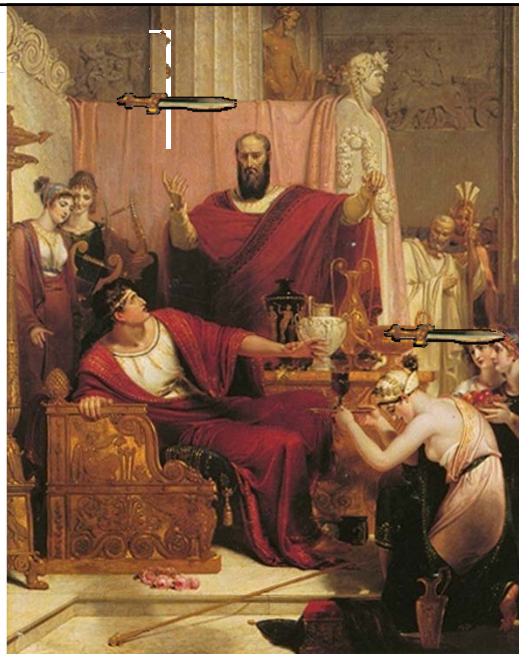
**$x.f(args)$**

Semantics: apply the  
feature  $f$ , with given  
 $args$  if any, to the  
object to which  $x$  is  
attached

... and basic issue  
studied here:

How do we guarantee  
that  $x$  will always be  
"attached" to an object?

(If not, call produces an exception and usually termination)



158

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. I was designing the first comprehensive type system for references in an object-oriented language (ALGOL W). My goal was to ensure that all use of references should be safe, checked by the compiler.

But I couldn't resist the temptation to put in a null reference, because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.



**Tony Hoare, Inventor of QuickSort, Turing Award Winner**



Sir Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, probably best known for the development in 1960 of Quicksort (or Hoaresort), one of the world's most widely used sorting algorithms.

He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP) used to specify the interactions of concurrent processes (including the Dining philosophers problem) and the inspiration for the Occam programming language.

159

## Plan



1. Context
2. New language constructs
3. Achieving void safety
4. Current status

160





- 1 -

# Context

161



Source: Patrice Chalin

44% of Eiffel preconditions clauses are of the form

$x \neq \text{Void}$

162

## Requirements



- Minimal language extension
- Statically, completely void safe
- Simple for programmer, no mysterious rules
- Reasonably simple for compiler
- Handles genericity
- Doesn't limit expressiveness
- Compatibility or minimum change for existing code
- 1<sup>st</sup>-semester teachability

163

## Lessons from Spec# work



*"Spec# stipulates the inference of non-[voidness] for local variables. This inference is performed as a dataflow analysis by the Spec# compiler."*

(Barnett, Leino, Schulte, Spec# paper)

*x != Void*

164

**Subject: "I had a dream"**

---



From:"Eric Bezault" [ericb@gobosoft.com](mailto:ericb@gobosoft.com)

To:"ECMA TC49-TG4" Date:Thu, 4 Jun 2009 11:21

Last night I had a dream. I was programming in Eiffel 5.7. The code was elegant. There was no need for defensive programming just by taking full advantage of design by contract. Thanks to these contracts the code was easy to reuse and to debug. I could hardly remember the last time I had a call-on-void-target. It was so pleasant to program with such a wonderful language.

This morning when I woke up I looked at the code that had been modified to comply with void-safety. This was a rude awakening. The code which was so elegant in my dream now looked convoluted, hard to follow. It looks like assertions are losing all their power and defensive programming is inviting itself again in the code. [...]

165

- 2 -

**New language  
constructs**

166

## New constructs

### 1. Object test

Replaces all "downcasting" (type narrowing) mechanisms

### 2. Type annotations: "attached" and "detachable"

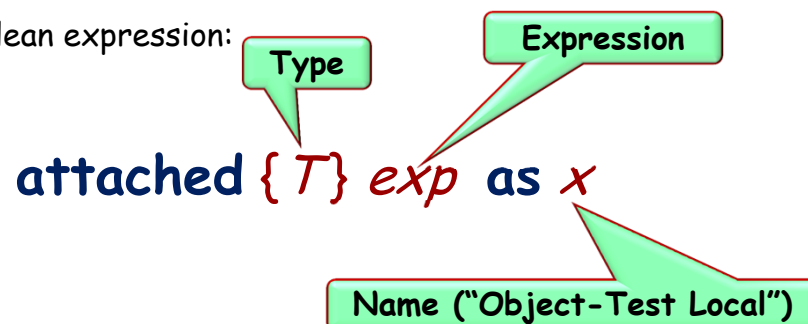
New keywords: **attached**, **detachable**

(Plus: **stable**.)

167

## The Object Test (full form)

Boolean expression:



Value:

True if value of *exp* is attached to an object of type *T* or conforming

Plus: binds *x* to that value over **scope** of object test

168

## Object Test example

if **attached { T } exp as x** then

... Arbitrary instructions...

**x**.operation

... Other instructions ...

end

Scope of **x**

169

## Object Test variants

**attached { T } exp as x**

**attached exp as x**

**attached { T } exp**

**attached exp**

Same semantics as  
**exp != Void**

170

## Another example of Object Test scope

```
from
...
until not attached exp as x loop

  ... Arbitrary instructions ...
  x.some_operation
  ... Other instructions ...

end
```

← Scope of *x*

171

## Object test in contracts

```
my_routine

  require
    attached exp as x and then x.some_property
  do
    ...
  end
```

↑  
Scope of *x*

172

- 3 -

## Achieving void safety

173

### A success story: static type checking

We allow

$x.f(args)$

What if  $x$  is void?

only if we can guarantee that at run time:

The object attached to  $x$ , **if it exists**, has a feature for  $f$ , able to handle the  $args$

Basic ideas:

- Accept it only if type of  $x$  has a feature  $f$
- Assignment  $x := y$  requires conformance (based on inheritance)

174

## Generalizing static type checking



The goal ("void safety"): at compile time, allow

*$x.f(args)$*

only if we can guarantee that at run time:

*$x$*  is not void

175