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!

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
- Supports full concurrency

**Environment** (EiffelStudio):
- Integrated, provides single solution, including analysis and modeling
- Lots of platforms (Unix, Windows, VMS, .NET...)
- Open and interoperable
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...
Eiffel is not...

Model-driven development
Functional programming
DSLs
Use-case-driven design

Designing from use cases
The competition

Rational Rose

SAP

SPARK

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

www.ecma-international.org/publications/standards/Ecma-367.htm
Attached types, conversion, assigner commands...
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

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**
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

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
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

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
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

Style of Eiffel language description

Specification on three levels:

> Syntax
> Validity
> Semantics
Syntax: structure of texts

Syntactically illegal examples:

\[ x.a = b \]
Syntax specification

Validity: constraints on syntactically legal texts

Invalid example:

your_integer + your_boolean
Semantics: effect of valid texts, if defined

Incorrect example:

\[ x := \text{Void} \]

\[ x.your\_feature \]

Validity rules: if and only if

**Local Entity rule**

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.
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.

The Eiffel language: there is a hidden agenda

That you forget it even exists
The Eiffel method

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...
### 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

### 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
Seamlessness

**Seamlessness Principle**

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

Reversibility

**Reversibility Principle**

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

Example classes:
- PLANE, ACCOUNT, TRANSACTION...
- STATE, COMMAND...
- HASH_TABLE...
- TEST_DRIVER...
- TABLE...

Analysis classes

defered 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
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

---

The seamless, reversible model

Diagram showing the seamless, reversible model with layers for Analysis, Design, Implementation, V&V, Generalization.
Generalization

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

The cluster model

Mix of sequential and concurrent engineering

Permits dynamic reconfiguration
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
- ...

EiffelStudio diagram tool
Text-graphics equivalence

Equivalence Principle
Textual, graphical and other views should all represent the same model
Command-Query separation principle

Asking a question should not change the answer

A command
A query

Asking a question should not change the answer

Command-Query separation principle
**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.

**Command-Query Separation**

**Asking a question should not change the answer!**
**Referential transparency**

If two expressions have equal value, one may be substituted for the other in any context where that other is valid.

If \( a = b \), then \( f(a) = f(b) \) for any \( f \).
Prohibits functions with side effects.

Also:
- For any integer \( i \), normally \( i + i = 2 \times i \)
- But even if \( \text{getint}() = 2 \), \( \text{getint}() + \text{getint}() \) is usually not equal to 4.

**Command-query separation**

Input mechanism using EiffelBase
(instead of \( n := \text{getint}() \)):

\[
\text{io.read_integer}
\]

\[
n := \text{io.last_integer}
\]
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

Language style

Compatibility principle

Traditional notations should be supported with an O-O semantics
Infix and prefix operators

In

\[ a - b \]

the \(-\) operator is "infix"

(written between operands)

In

\[ -b \]

the \(-\) operator is "prefix"

(written before the operand)

The object-oriented form of call

\[ \textit{some\_target} \textit{.some\_feature} (\textit{some\_arguments}) \]

For example:

\[ \textit{my\_figure.display} \]
\[ \textit{my\_figure.move (3, 5)} \]
\[ x := a + b \]
Operator features

expanded class INTEGER feature

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

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

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

... end

Calls such as \( i.\text{plus}(j) \) can now be written \( i + j \).

Assignment commands

It is possible to define a query as

\[
\text{temperature: REAL assign set\_temperature}
\]

Then the syntax

\[
x.\text{temperature} := 21.5
\]

is accepted as an abbreviation for

\[
x.\text{set\_temperature}(21.5)
\]

Retains contracts and any other supplementary operations.
Using the bracket alias

In class ARRAY[G]:

```
item alias "[]" (i : INTEGER): G
require
    i >= lower and i <= count
do ... end
```

```
put (x: G; i: INTEGER): G
require
    i >= lower and i <= count
do ... end
```

```
Not an assignment!
```

```
a.put (a.item (i) + 1, i)
a.item (i) := a.item (i) + 1
```

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

Bracket alias

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

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

Object-oriented forms:

\[
\begin{align*}
  a & : ARRAY[T] \\
  a \cdot put(x, 23) \\
  x & := a \cdot item(23)
\end{align*}
\]

Above mechanisms make the following synonyms possible:

\[
\begin{align*}
  a[23] & := x \\
  x & := a[23]
\end{align*}
\]

Usual form:

\[
\begin{align*}
  a[i] & := a[i] + 1
\end{align*}
\]

Object-oriented form:

\[
\begin{align*}
  a \cdot put(a \cdot item(i) + 1, i)
\end{align*}
\]

Design by Contract
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.
**Design by Contract: applications**

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

**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
A counter-example: Ariane 5, 1996


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!

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 detected by either static inspection or testing:

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

Contracts for analysis

<table>
<thead>
<tr>
<th>fill</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>(Satisfy precondition:) Make sure input valve is open, output valve closed</td>
<td>(From postcondition:) Get filled-up tank, with both valves closed</td>
</tr>
<tr>
<td>Supplier</td>
<td>(Satisfy postcondition:) Fill the tank and close both valves</td>
<td>(From precondition:) Simpler processing thanks to assumption that valves are in the proper initial position</td>
</tr>
</tbody>
</table>
### Constracts for analysis

defferred class VAT
  inherit TANK
  feature
    in_valve, out_valve: VALVE
    fill
      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

### 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
end
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

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
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

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
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

The contract

<table>
<thead>
<tr>
<th>withdraw</th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Make sure sum is neither too small nor too big</td>
<td>(From postcondition:) Get account updated with sum withdrawn</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Update account for withdrawal of sum</td>
<td>(From precondition:) Simpler processing: may assume sum is within allowable bounds</td>
</tr>
</tbody>
</table>
The imperative and the applicative

<table>
<thead>
<tr>
<th>do</th>
<th>ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance := balance - sum</td>
<td>balance = old balance - sum</td>
</tr>
<tr>
<td>PRESCRIPTIVE</td>
<td>DESCRIPTIVE</td>
</tr>
</tbody>
</table>

How? | What?  
Operational | Denotational  
Implementation | Specification 
Command | Query  
Instruction | Expression  
Imperative | Applicative

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
The correctness of a class

For every creation procedure \( cp \):

\[
\{ \text{Pre}_{cp} \} \text{ do } \{ \text{INV and Post}_{cp} \}
\]

For every exported routine \( r \):

\[
\{ \text{INV and Pre}_{r} \} \text{ do } \{ \text{INV and Post}_{r} \}
\]
Extending the basic notion of class

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:

```plaintext
What if wrong?
c : CITY; p : PERSON
cities : LIST ...
people : LIST ...
---------------------------------------------------------
people.extend (p)
cities.extend (c)
c := cities.last
c. some_city_operation
```
A generic class

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

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

    cities: LIST[CITY]

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)
**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*.

**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.)
Adding two vectors

\[ u + v = w \]

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
Constrained genericity

Body of \textit{plus alias} "+":

\begin{verbatim}
create Result.make(lower, upper)
from
  i := lower
until
  i > upper
loop
  a := item(i)
  b := other.item(i)
  \textcolor{green}{c := a + b} \quad \text{-- Requires "+" operation on } G!
  Result.put(c, i)
  i := i + 1
end
\end{verbatim}

The solution

Declare class \textit{VECTOR} as

\begin{verbatim}
class VECTOR[G \rightarrow NUMERIC] feature
  \ldots \text{The rest as before} \ldots
end
\end{verbatim}

Class \textit{NUMERIC} (from the Kernel Library) provides
features \textit{plus alias} "+", \textit{minus alias} "-" and so on.
Improving the solution

Make \textit{VECTOR} itself a descendant of \textit{NUMERIC}, effecting the corresponding features:

\begin{verbatim}
class VECTOR [G -> NUMERIC] inherit NUMERIC

feature
  ... Rest as before, including infix "+"...
end
\end{verbatim}

Then it is possible to define

\begin{verbatim}
v : VECTOR [INTEGER]
vv : VECTOR [VECTOR [INTEGER]]
vvv : VECTOR [VECTOR [VECTOR [INTEGER]]]
\end{verbatim}

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: \texttt{x.f(...)}
The correctness of a class

For every creation procedure \( cp \):

\[
\{ \text{Pre}_{cp} \} \text{ do } \{ \text{INV and Post}_{cp} \}
\]

For every exported routine \( r \):

\[
\{ \text{INV and Pre}_{r} \} \text{ do } \{ \text{INV and Post}_{r} \}
\]

Uniform Access

\(\text{(A1)}\)

\[
\begin{align*}
\text{list_of_deposits} & : 200 & 100 & 500 & 1000 \\
\text{list_of_withdrawals} & : 800 & 100 & 100
\end{align*}
\]

\(\text{(A2)}\)

\[
\begin{align*}
\text{list_of_deposits} & : 200 & 300 & 500 & 1000 \\
\text{list_of_withdrawals} & : 800 & 100 & 100 \\
\text{balance} & : 1000
\end{align*}
\]

\[\text{balance} = \text{deposits.total} - \text{withdrawals.total}\]

\[a : \text{ACCOUNT}\]

\[\text{...}\]

\[\text{print(a.balance)}\]
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

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”)
Contracts and quality assurance

Precondition violation: Bug in the client.
Postcondition violation: Bug in the supplier.
Invariant violation: Bug in the supplier.

\{P\} \ A \ \{Q\}

Contracts: run-time effect

Compilation options (per class, in Eiffel):
- No assertion checking
- Preconditions only
- Preconditions and postconditions
- Preconditions, postconditions, class invariants
- All assertions
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

Lists in EiffelBase
Trying to insert too far right

```
1
```

Cursor

(Already past last element!)

A command and its contract

```
put_right (v: like item) is
    -- Add 'v' to the right of cursor position.
    -- Do not move cursor.
    do
        p := new_cell (v)
        count := count + 1
    ensure -- from DYNAMIC_CHAIN
        new_count: count = old count + 1
        same_index: index = old_index
    require -- from DYNAMIC_CHAIN
        extendible: extendible
        not_after: not after
```

Precondition

Postcondition
Moving the cursor forward

Two queries, and command `forth`

```plaintext
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
```
Where the cursor may go

Valid cursor positions

From the invariant of class \textit{LIST}

\begin{verbatim}
invariant

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

-- from \textit{CHAIN}

  non_negative_index: index >= 0
  index_small_enough: index <= count + 1
\end{verbatim}

Valid cursor positions
Contracts and bug types

Preconditions are particularly useful to find bugs in client code:

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

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

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
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

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
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

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?)
The next step

Proofs

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.
Uses of the contract & interface forms

- Documentation, manuals
- Design
- Communication between developers
- Communication between developers and managers

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 interface forms.

Contracts and inheritance

Correct call in C:

\[\text{if } a_1.\alpha \text{ then }\]
\[a_1.r(\ldots)\]

-- Here \(a_1.\beta\) holds

Client \rightarrow \text{Inheritance} \quad \text{++ Redefinition}
Assertion redeclaration rule

When redeclaring a routine, we may only:

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

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`
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.

The original strategy

```plaintext
r(...) is
  require
    ...
  do
    op_1
    op_2
    ...
    op_i
    ...
    op_n
  ensure
    ...
end
```
Not going according to plan

\[ r(...) \text{ is require ... do } op_1 \]
\[ op_2 \]
\[ ... \]
\[ op_i \]
\[ ... \]
\[ end \]

Fails, triggering an exception in \( r \) (\( r \) is recipient of exception).

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)
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).

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
Transmitting over an unreliable line (2)

\[ \text{Max_attempts: INTEGER } = 100 \]
\[ \text{failed: BOOLEAN} \]

\[ \text{attempt_transmission(message: STRING)} \]
\[ \quad \text{-- Try to transmit message,} \]
\[ \quad \text{-- if impossible in at most \text{Max_attempts}} \]
\[ \quad \text{-- attempts, set \text{failed} to true.} \]

\[ \text{local} \]
\[ \text{failures: INTEGER} \]
\[ \text{do} \]
\[ \text{if failures < Max_attempts then} \]
\[ \quad \text{unsafe_transmit(message)} \]
\[ \text{else} \]
\[ \quad \text{failed := True} \]
\[ \text{end} \]
\[ \text{rescue} \]
\[ \quad \text{failures := failures + 1} \]
\[ \text{retry} \]

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!
Eiffel Model Library (MML)

Bernd Schoeller, Tobias Widmer, Nadia Polikarpova

Classes correspond to mathematical concepts:

\[ \text{SET}[G], \text{FUNCTION}[G, H], \text{TOTAL_FUNCTION}[G, H], \text{RELATION}[G, H], \text{SEQUENCE}[G], \ldots \]

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

Specified with contracts (unproven) reflecting mathematical properties

Expressed entirely in Eiffel

---

Specifying lists

class \text{LINKED_LIST}[G]

feature

... remove_front

-- Remove first item.

require not empty

do

\text{first} := \text{first}.right

ensure \text{model} = \text{old model}.tail
\text{count} = \text{old count} - 1

\text{first} = \text{old item (2)}

end
Example MML class

```mml
class SEQUENCE[G] feature
  count : NATURAL  -- Number of items
  last : G       -- Last item

  extended (x) : SEQUENCE[G]
    ensure
    -- Identical sequence except x added at end.
    Result.count = count + 1
    Result.last = x
    Result.sub(1, count) ~ Current

  mirrored : SEQUENCE[G]
    ensure
    -- Same items in reverse order.
    Result.count = count
```

Principles

Very simple mathematics only
  - Logic
  - Set theory
EiffelBase+

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

Verification As a Matter Of Course
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

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...
Concurrency in Eiffel: SCOOP

No data races
Concurrency in Eiffel: SCOOP

No data races
Concurrency in Eiffel: SCOOP

No data races
Concurrency in Eiffel: SCOOP

No data races
Concurrency in Eiffel: SCOOP

No data races
Concurrency in Eiffel: SCOOP

No data races
Concurrency in Eiffel: SCOOP

No data races
Concurrency in Eiffel: SCOOP

No data races
Concurrency in Eiffel: SCOOP

No data races
No data races
**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)
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.

Plan

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

Source: Patrice Chalin

44% of Eiffel preconditions clauses are of the form

\[ x \neq \text{Void} \]
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
- 1st-semester teachability

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)

\[ x /= \text{Void} \]
Subject: “I had a dream”

From: "Eric Bezault" 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. […]

- 2 -

New language constructs
New constructs

1. Object test

   Replaces all "downcasting" (type narrowing) mechanisms

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

New keywords: attached, detachable

(Plus: stable.)

The Object Test (full form)

Boolean expression:

\[
\text{attached } \{ \mathcal{T} \} \exp \text{ as } x
\]

Value:

True if value of \( \exp \) is attached to an object of type \( \mathcal{T} \) or conforming

Plus: binds \( x \) to that value over scope of object test
Object Test example

if \( \text{attached } \{ T \} \ exp \ \text{as } x \) then

... Arbitrary instructions...

\( x \).operation

... Other instructions ...

end

Scope of \( x \)


Object Test variants

\text{attached } \{ T \} \ exp \ \text{as } x

\text{attached } \ exp \ \text{as } x

\text{attached } \{ T \} \ exp

\text{attached } \ exp

Same semantics as \( exp \neq \text{Void} \)
Another example of Object Test scope

from ...
until not attached exp as x loop
  ... Arbitrary instructions ...
  x.some_operation
  ... Other instructions ...
end

Object test in contracts

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

Scope of x
Achieving void safety

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)
Generalizing static type checking

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

\[ x.f(args) \]

only if we can guarantee that at run time:

\[ x \text{ is not void} \]