ETH Zürich

Chair of Software Engineering

Software Architecture

Bertrand Meyer, Carlo A. Furia, Martin Nordio

ETH Zurich, February-May 2011

Lecture 15: Design by Contract and exception handling

Topics

Part 1: Key concepts

Part 2: Contracts & documentation

Part 3: Contracts & testing

Part 4: Contracts & analysis, methodological notes

Part 5: Contracts & inheritance

Part 6: Contracts & loops

Part 7: Handling abnormal cases

Part 8: Contracts in various languages

Part 9: New developments

Part 10: Conclusion

- 1 -

Key concepts

Design by Contract

A discipline of analysis, design, implementation, management

Applications throughout the software lifecycle:

- Getting the software right: analysis, design , implementation
- Debugging & testing
- Automatic documentation
- Getting inheritance right
- Getting exception handling right
- Maintenance
- Management

Background

Work on “axiomatic semantics”:

- R.W. Floyd (1967)
- C.A.R. Hoare (1969, 1972)
- E.W. Dijkstra (1978)

1970’s languages: CLU, Alphard

Eiffel (from 1985): connection with object technology

90s and onward: contract additions to numerous languages: C++, Java, C#, UML

Design by Contract

Every software element is intended to satisfy a certain goal, or *contract*

for the benefit of other software elements (and ultimately of human users)

The contract of any software element should be

- Explicit
- Part of the software element itself

The three questions

- What does it expect?

Precondition

- What does it promise?

Postcondition

- What does it maintain?

Class invariant

Contracting components

Definition of what each element of the functionality:

- Expects (*precondition*)
- Promises (*postcondition*)
- Maintains (*invariant*)

Does not have to be complete (but wait)

What we do with contracts

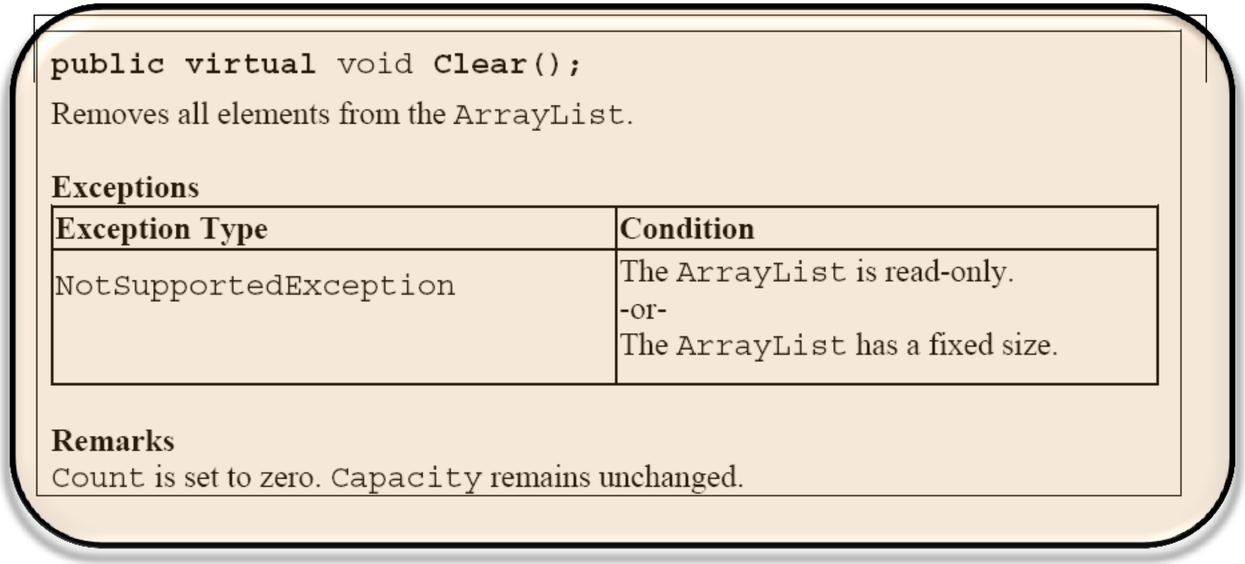
Write better software
Analyze
Design
Reuse
Implement
Use inheritance properly
Avoid bugs
Document software automatically
Help project managers do their job

(with run-time monitoring)

Perform systematic testing
Guide the debugging process

With and without contracts

with Karine Arnout (IEEE Computer)



`public virtual void Clear();`

Removes all elements from the `ArrayList`.

Exceptions

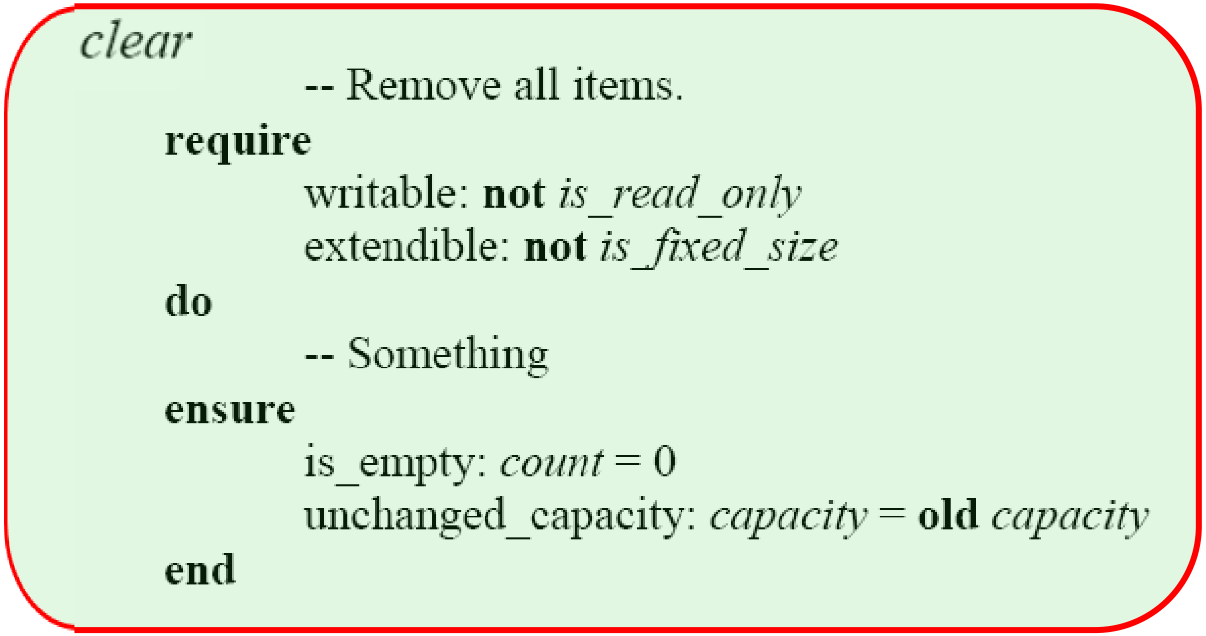
Exception Type	Condition
`NotSupportedException`	The `ArrayList` is read-only. -or- The `ArrayList` has a fixed size.

Remarks

`Count` is set to zero. `Capacity` remains unchanged.

.Net collections library

EiffelBase



```
clear
            -- Remove all items.
      require
            writable: not is_read_only
            extendible: not is_fixed_size
      do
            -- Something
      ensure
            is_empty: count = 0
            unchanged_capacity: capacity = old capacity
      end
```

The underlying view

Software construction consists of building 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

Correctness in software

Correctness is a relative notion: consistency of implementation vis-à-vis specification.

Basic notation: (*P*, *Q* : assertions, i.e. properties of the state of the computation. *A* : instructions).

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

"Hoare triple"

What this means (total correctness):

- Any execution of *A* started in a state satisfying *P* will terminate in a state satisfying *Q*.

Hoare triples: a simple example

$\{n > 5\}\ n := n + 9\ \{n > 13\}$

Most interesting properties:

- *Strongest* postcondition (from given precondition).
- *Weakest* precondition (from given postcondition).

"*P* is stronger than or equal to *Q*" means:

P **implies** *Q*

QUIZ: What is the strongest possible assertion? The weakest?

A contract (from EiffelBase)

```
extend (new : G; key : H)
                -- Assuming there is no item of key key,
                -- insert new with key ; set inserted.
        require
                key_not_present: not has (key)
        ensure
                insertion_done: item (key) = new
                key_present: has (key)
                inserted: inserted
                one_more: count = old count + 1
```

Software correctness (another quiz)

Consider

{*P*} *A* {*Q*}

Take this as a job ad in the classifieds

Should a lazy employment candidate hope for a weak or strong *P*? What about *Q*?

Two "special offers":

- 1. {*False*} *A* {...}
- 2. {...} *A* {*True*}

Properties of human contracts

A contract:

- Binds two parties (or more): supplier, client
- Is explicit (written)
- Specifies mutual obligations and benefits
- Usually maps obligation for one of the parties into benefit for the other, and conversely
- Has no hidden clauses: obligations are those specified
- Often relies, implicitly or explicitly, on general rules applicable to all contracts: laws, regulations, standard practices

A human contract

deliver	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Bring package before 4 p.m.; pay fee.	(From postcondition:) Get package delivered by 10 a.m. next day.
Supplier	(Satisfy postcondition:) Deliver package by 10 a.m. next day.	(From precondition:) Not required to do anything if package delivered after 4 p.m., or fee not paid.

Properties of human contracts

A contract:

- Binds two parties (or more): supplier, client
- Is explicit (written)
- Specifies mutual obligations and benefits
- Usually maps obligation for one of the parties into benefit for the other, and conversely
- Has no hidden clauses: obligations are those specified
- Often relies, implicitly or explicitly, on general rules applicable to all contracts: laws, regulations, standard practices

Contracts for analysis, specification

```
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, ... [Other features] ...

invariant

      is_full = (gauge >= 0.97 * maximum)  and  (gauge <= 1.03 * maximum)

end
```

Precondition

Specified, but not implemented

Postcondition

Class invariant

Contracts for analysis

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

“So, it’s like assert.h?”

Source: Reto Kramer

Design by Contract goes further:

- “Assert” does not provide a contract
- Clients cannot see asserts as part of the interface
- Asserts do not have associated semantic specifications
- Not explicit whether an assert represents a precondition, post-conditions or invariant
- Asserts do not support inheritance
- Asserts do not yield automatic documentation

A class without contracts

```
class
        ACCOUNT
feature -- Access

        balance : INTEGER
                -- Balance

        Minimum_balance: INTEGER = 1000
                -- Lowest permitted balance

feature {NONE} -- Deposit and withdrawal

        add (sum : INTEGER)
                        -- Add sum to the balance.
                do
                        balance := balance + sum
                end
```

Secret features

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
```

Value returned by function

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
```

Precondition

Postcondition

Introducing contracts

Precondition

```
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
```

Postcondition

Value of *balance*, captured on entry to routine

The imperative and the applicative

do *balance* := *balance* - *sum*	**ensure** *balance* = **old** *balance* - *sum*
PRESCRIPTIVE	DESCRIPTIVE
How?	What?
Operational	Denotational
Implementation	Specification
Command	Query
Instruction	Expression
Imperative	Applicative

The contract

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

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 class invariant

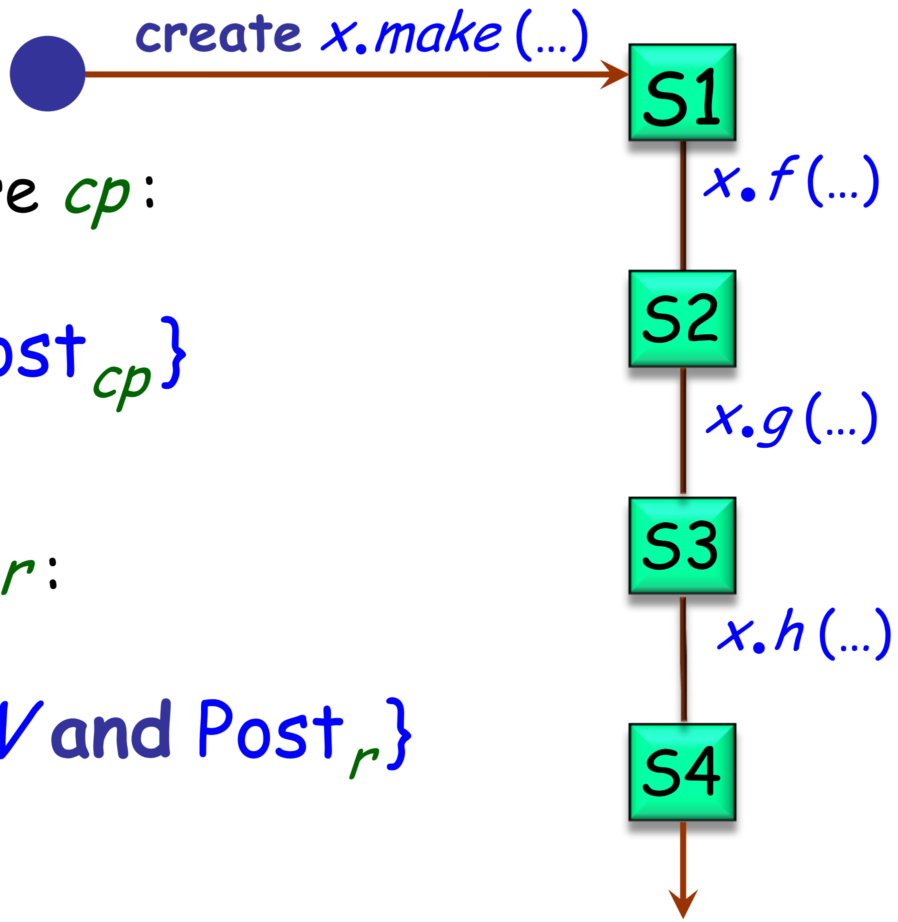
Consistency constraint applicable to all instances of a class.

Must be satisfied:

- After creation
- After execution of any feature by any client

  Qualified calls only: *x*.*f* (...)

The correctness of a class



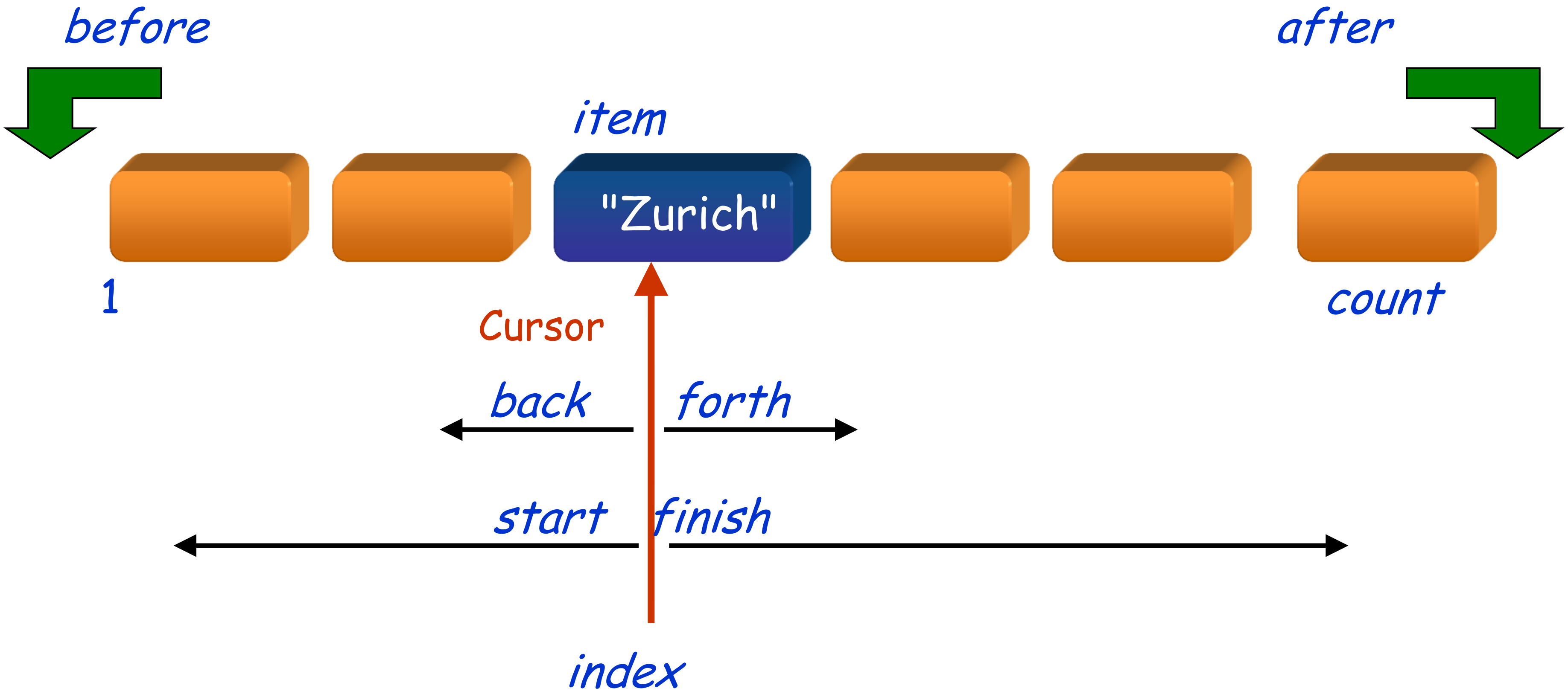
For every creation procedure *cp* :

$\{Pre_{cp}\}\ \mathbf{do}_{cp}\ \{INV \text{ and } Post_{cp}\}$

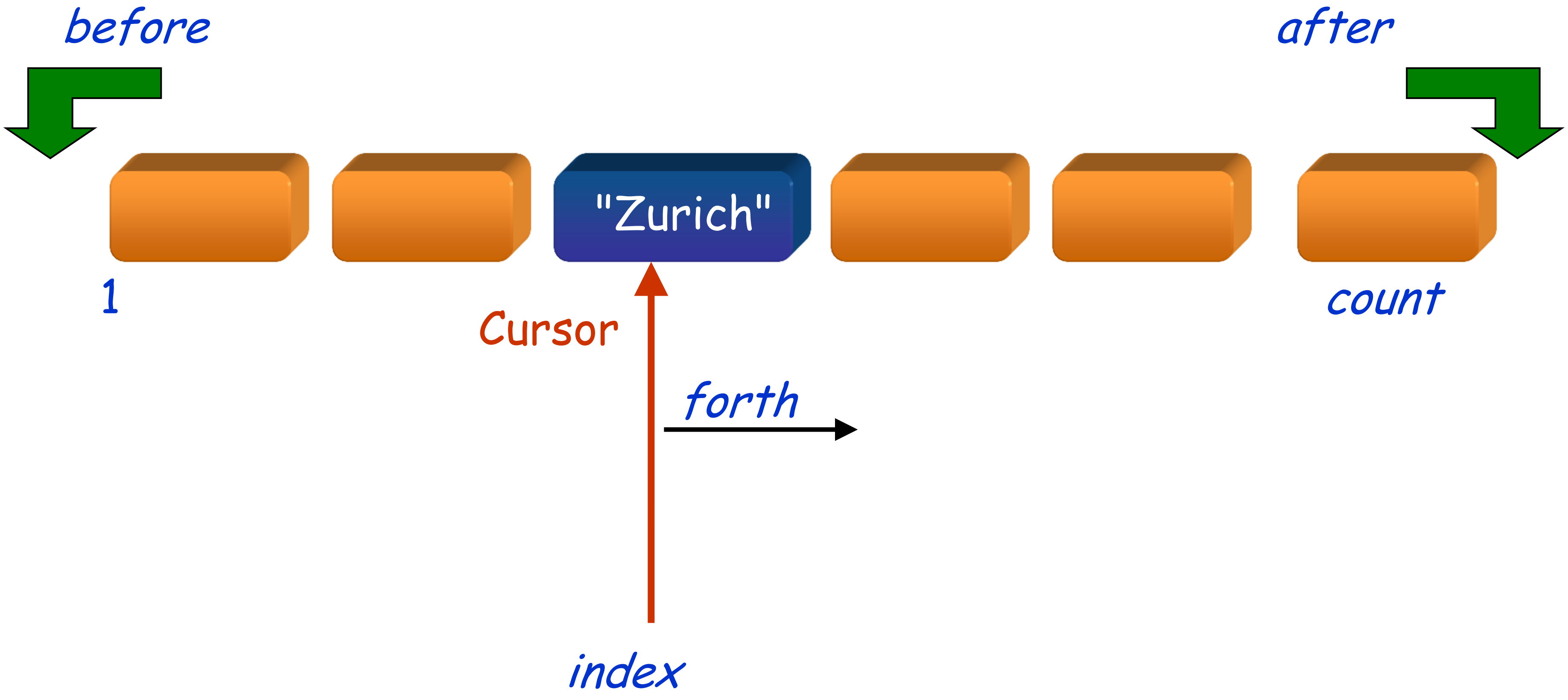
For every exported routine *r* :

$\{INV \text{ and } Pre_{r}\}\ \mathbf{do}_{r}\ \{INV \text{ and } Post_{r}\}$

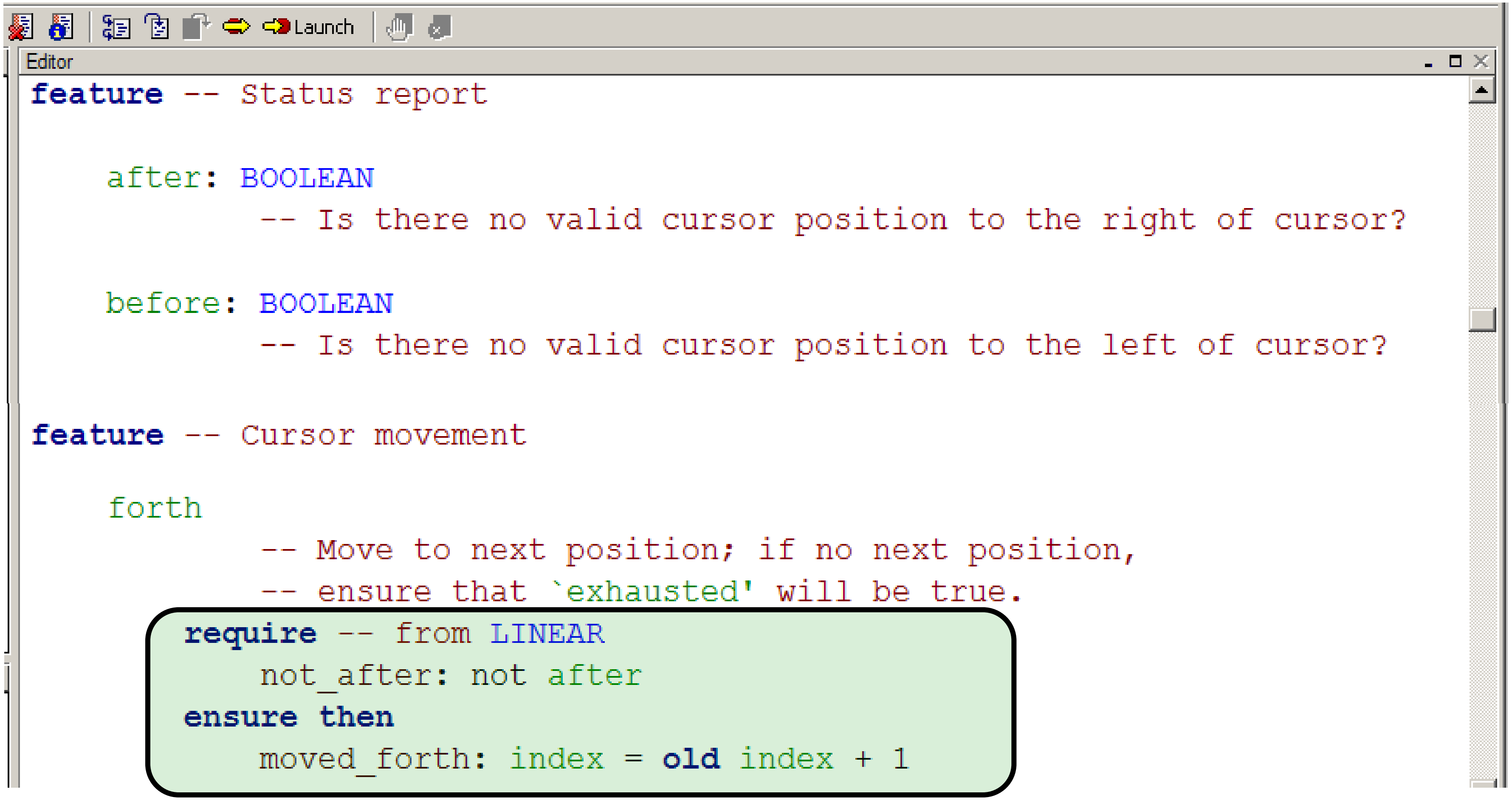
Lists in EiffelBase



Moving the cursor forward



Two queries, and command *forth*



```
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
```

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)
        -- Push x on top of stack.
    require
        not is_full
    do
        ...
    ensure
        not is_empty
    end
```

```
remove
        -- Pop top of stack.
    require
        not is_empty
    do
        ...
    ensure
        not is_full
    end
```

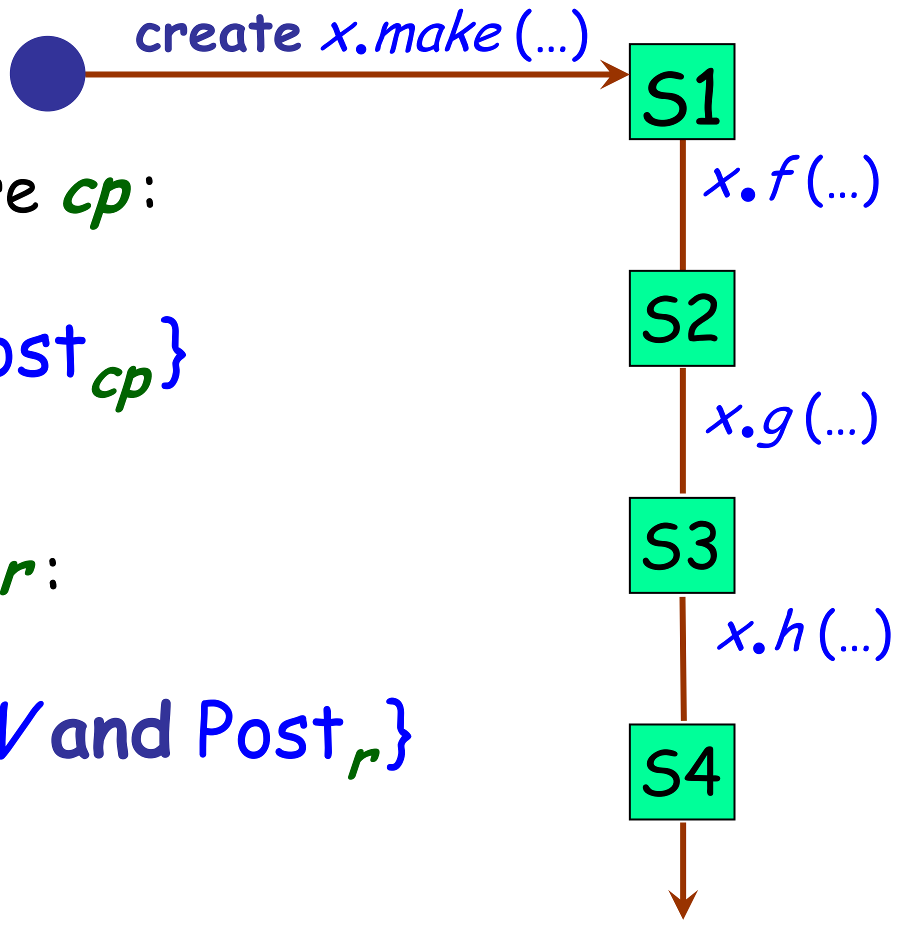
The correctness of a class

For every creation procedure *cp*:

$\{Pre_{cp}\}$ **do**$_{cp}$ $\{INV \text{ and } Post_{cp}\}$

For every exported routine *r*:

$\{INV \text{ and } Pre_{r}\}$ **do**$_{r}$ $\{INV \text{ and } Post_{r}\}$



create *x.make* (...)

S1

x.f (...)

S2

x.g (...)

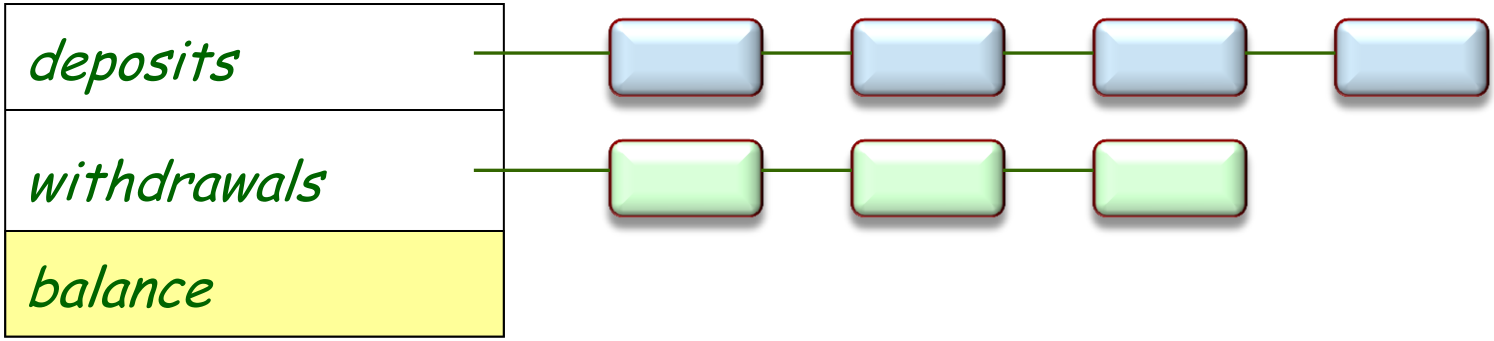
S3

x.h (...)

S4

A slightly more sophisticated version

(A2)



balance = *deposits*.*total* - *withdrawals*.*total*

New version

```
class
    ACCOUNT
create
    make
feature {NONE} – Implementation

    add (sum : INTEGER)
            -- Add sum to the balance.
        do
            balance := balance + sum
        ensure
            balance_increased: balance = old balance + sum
        end

      deposits : DEPOSIT_LIST

      withdrawals : WITHDRAWAL_LIST
```

New version

```
feature {NONE} -- Initialization
    make (initial_amount: INTEGER)
                -- Set up account with initial_amount.
        require
            large_enough: initial_amount >= Minimum_balance
        do
            balance := initial_amount

            create deposits.make

            create withdrawals.make
        ensure
            balance_set: balance = initial_amount
        end
feature -- Access
        balance: INTEGER
            -- Balance
        Minimum_balance: INTEGER  =  1000
            -- Minimum balance
```

New version

```
feature -- Deposit and withdrawal operations

    deposit (sum : INTEGER)
            -- Deposit sum into the account.
        require
            not_too_small: sum >= 0
        do
            add (sum)

            deposits.extend (create {DEPOSIT}.make (sum))
        ensure
            increased: balance = old balance + sum

            one_more: deposits.count = old deposits.count + 1
        end
```

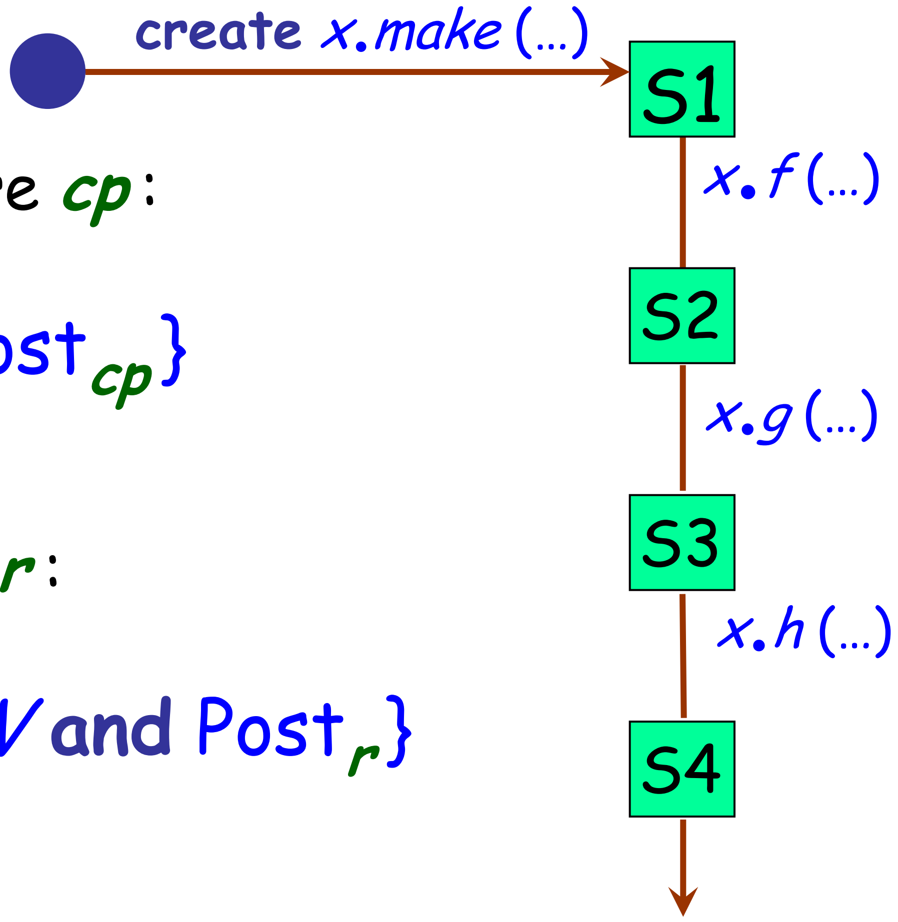
New version

```
withdraw (sum : INTEGER)
        -- Withdraw sum from the account.
    require
        not_too_small: sum >= 0
        not_too_big: sum <= balance - Minimum_balance
    do
        add (- sum)

        withdrawals.extend (create {WITHDRAWAL}.make (sum))

    ensure
        decreased: balance = old balance - sum
        one_more: withdrawals.count = old withdrawals.count + 1
    end
```

New version

```
    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

    consistent: balance = deposits.total - withdrawals.total

end
```

The correctness of a class



For every creation procedure *cp*:

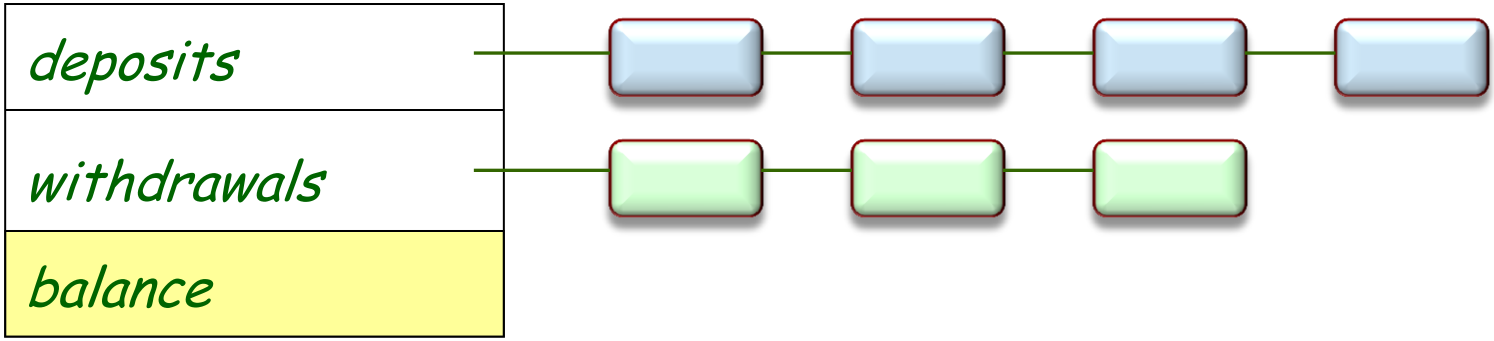
$$\{Pre_{cp}\}\ \mathbf{do}_{cp}\ \{INV \text{ and } Post_{cp}\}$$

For every exported routine *r*:

$$\{INV \text{ and } Pre_{r}\}\ \mathbf{do}_{r}\ \{INV \text{ and } Post_{r}\}$$

The new representation

(A2)



balance = *deposits*.*total* - *withdrawals*.*total*

Getting it right

```
feature {NONE} – Initialization

        make (initial_amount: INTEGER)
                        -- Set up account with initial_amount.
                require
                        large_enough: initial_amount >= Minimum_balance
                do
                        create deposits.make

                        create withdrawals.make

                        ~~balance := initial_amount~~

                        deposit (initial_amount)
                ensure
                        balance_set: balance = initial_amount
                end
```

What's wrong with this?

Design by contract: some applications

Getting the software right

Getting object-oriented development right: exceptions, inheritance…

Analysis and design

Automatic documentation

Project management

Maintenance

Testing and debugging

- 2 -

Contracts & documentation

Contracts for documentation

Contract view of a class: 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 code (reminder)

```
class
    ACCOUNT
create
    make
feature {NONE} – Implementation

    add (sum : INTEGER)
            -- Add sum to the balance.
        do
            balance := balance + sum
        ensure
            balance_increased: balance = old balance + sum
        end

        deposits : DEPOSIT_LIST

        withdrawals : WITHDRAWAL_LIST
```

The code (reminder)

```
feature {NONE} -- Initialization
	make (initial_amount: INTEGER)
				-- Set up account with initial_amount.
		require
			large_enough: initial_amount >= Minimum_balance
		do
			balance := initial_amount

			create deposits.make

			create withdrawals.make
		ensure
			balance_set: balance = initial_amount
		end
feature -- Access
		balance: INTEGER
			-- Balance
		Minimum_balance: INTEGER  =  1000
			-- Minimum balance
```

The code (reminder)

```
feature -- Deposit and withdrawal operations

    deposit (sum : INTEGER)
            -- Deposit sum into the account.
        require
            not_too_small: sum >= 0
        do
            add (sum)

            deposits.extend (create {DEPOSIT}.make (sum))
        ensure
            increased: balance = old balance + sum

            one_more: deposits.count = old deposits.count + 1
        end
```

The code (reminder)

```
withdraw (sum : INTEGER)
        -- Withdraw sum from the account.
    require
        not_too_small: sum >= 0
        not_too_big: sum <= balance - Minimum_balance
    do
        add (- sum)

        withdrawals.extend (create {WITHDRAWAL}.make (sum))


    ensure
        decreased: balance = old balance - sum
        one_more: withdrawals.count = old withdrawals.count + 1
    end
```

The code (reminder)

```
        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

        consistent: balance = deposits.total – withdrawals.total

end
```

Contract view

```
class interface ACCOUNT create
        make
feature
        balance: INTEGER
                -- Balance

        Minimum_balance : INTEGER = 1000
                -- Minimum balance

        deposit (sum: INTEGER)
                        -- Deposit sum into the account.
                require
                        not_too_small: sum >= 0
                ensure
                        increased: balance = old balance + sum
```

Contract view (continued)

```
    withdraw (sum: INTEGER)
                -- Withdraw sum from the account.
          require
                not_too_small: sum >= 0
                not_too_big: sum <= balance – Minimum_balance
          ensure
                decreased: balance = old balance – sum

    may_withdraw (sum: INTEGER): BOOLEAN
          -- Is it permitted to withdraw sum from the account?

invariant
    not_under_minimum: balance >= Minimum_balance
end
```

Documenting a program

Who will do the program documentation (technical writers, developers) ?

How to ensure that it doesn't diverge from the code (the reverse Dorian Gray syndrome) ?

The Single Product principle

The product is the software

Export rule for preconditions

In

```
feature
    r (...)
        require
            some_property
```

some_property must be exported!

No such requirement for postconditions and invariants.

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.

- 3 -

Contracts and testing

Contracts for testing

Contracts provide the right basis:

- A fault is a discrepancy between intent and reality
- Contracts describe intent

A contract violation always signals a fault:

- Precondition: in client
- Postcondition or invariant: in routine (supplier)

In EiffelStudio: select compilation option for contract monitoring at level of class, cluster or system.

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: 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

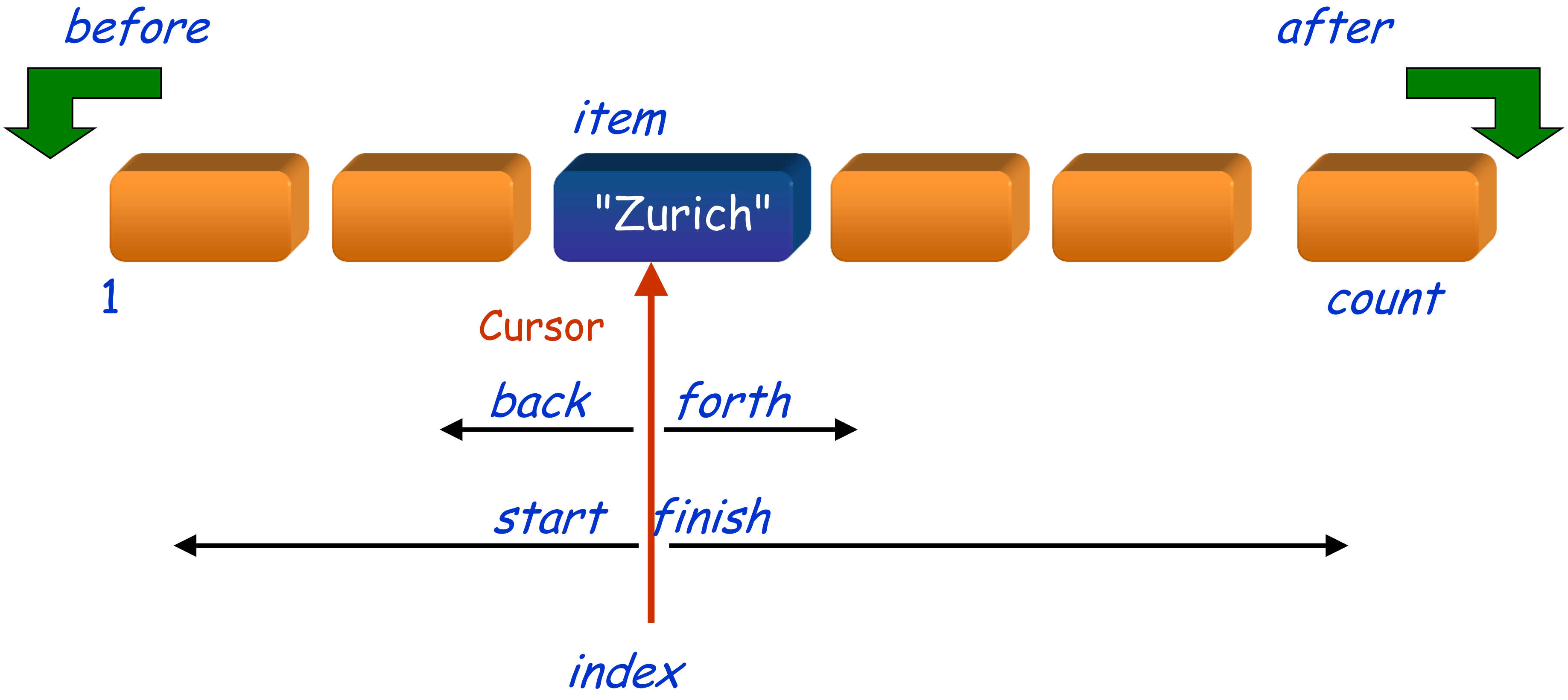
A revolutionary form of quality assurance

Contract monitoring

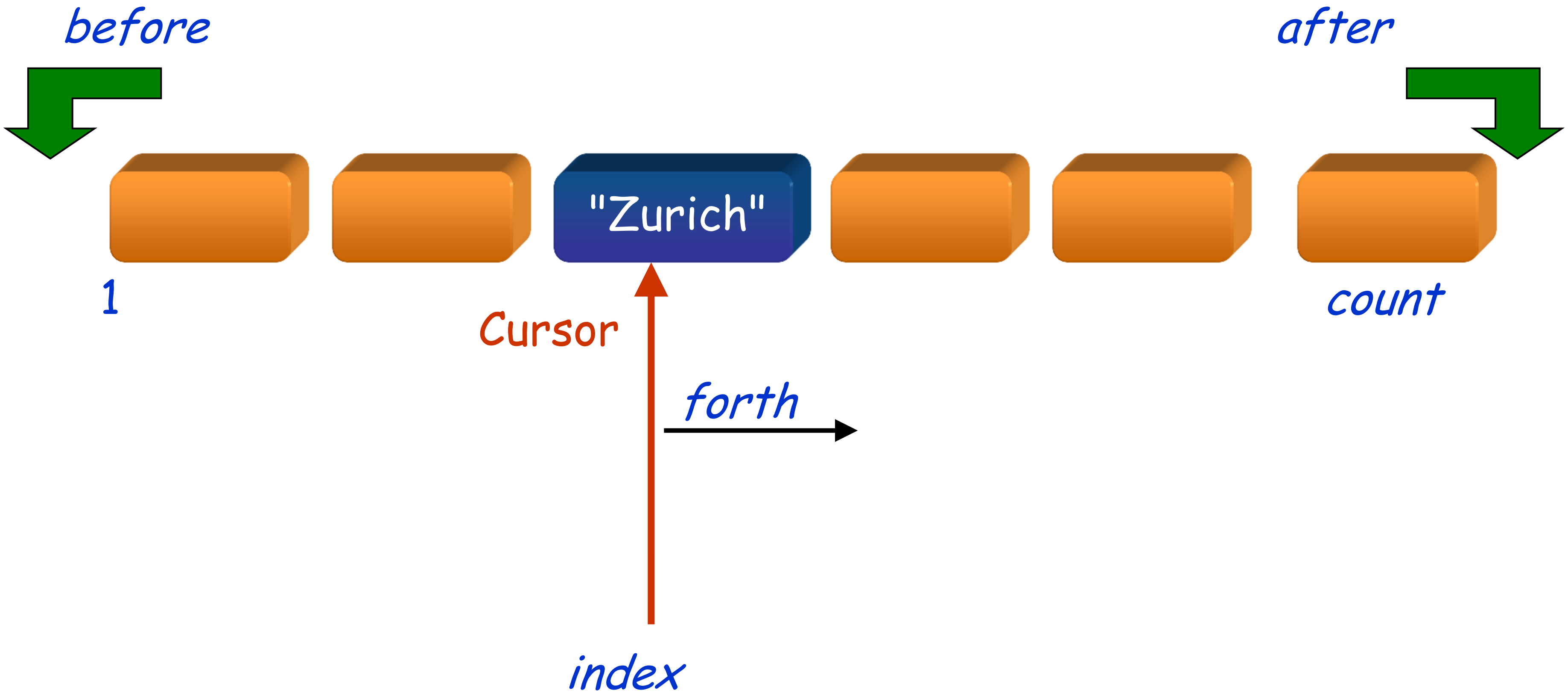
- Enabled or disabled by compile-time options.
- Default: preconditions only.
- In development: use "all assertions" whenever possible.
- During operation: normally, should disable monitoring. But have an assertion-monitoring version ready for shipping.
- Result of an assertion violation: exception.

Ideally: static checking (proofs) rather than dynamic monitoring.

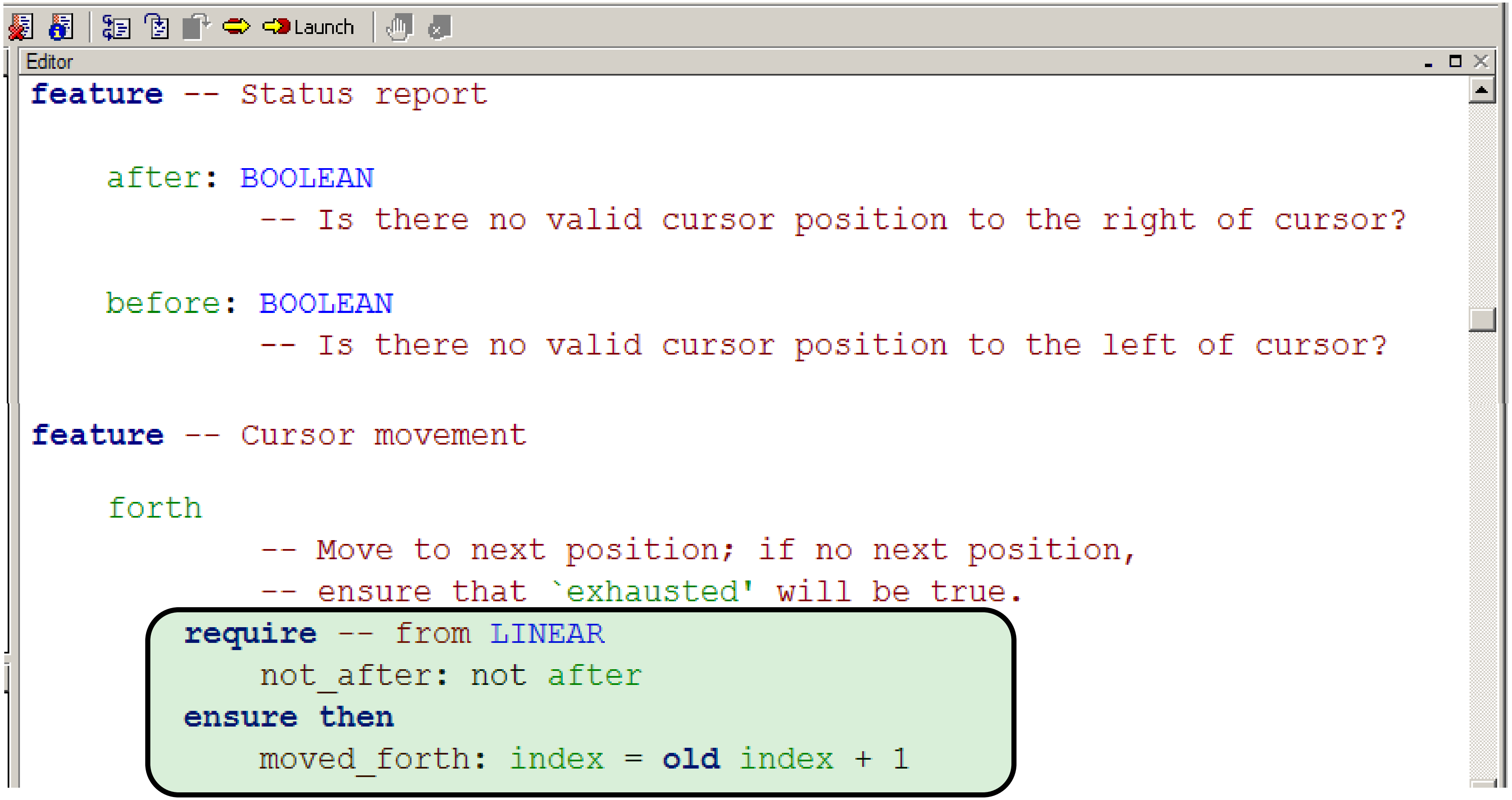
Lists in EiffelBase



Moving the cursor forward



Two queries, and command *forth*

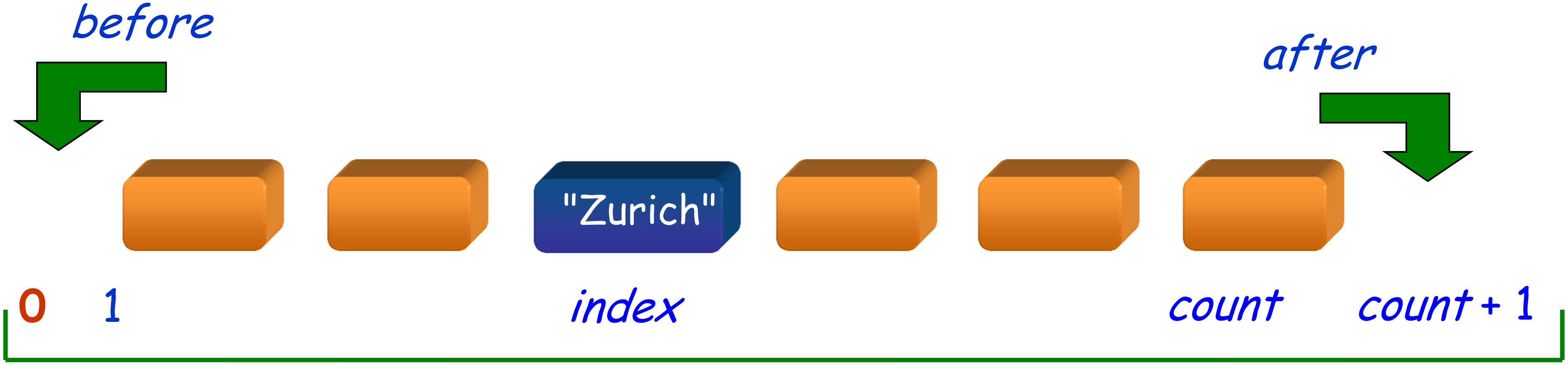


```
feature -- Status report

    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
```

Trying to insert too far right

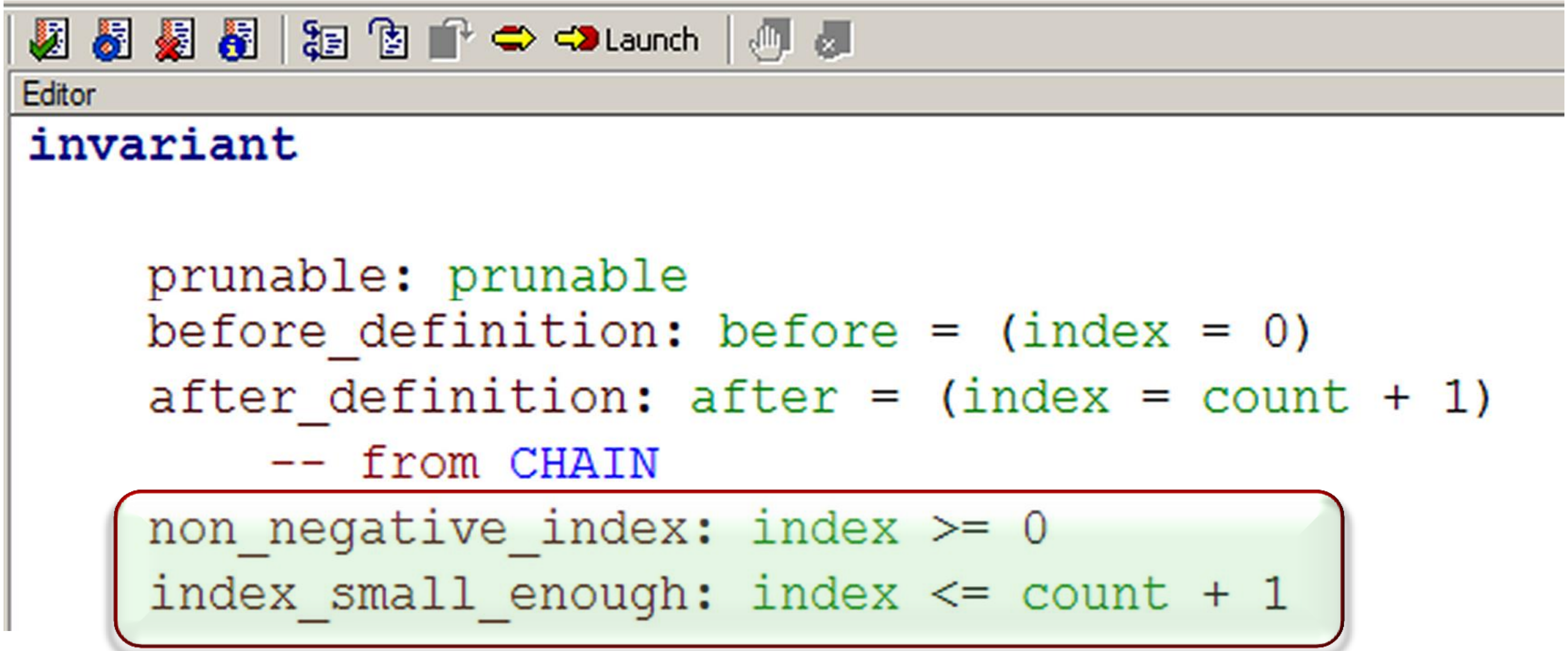
(Already past last element!)

Where the cursor may go



Valid cursor positions

From the invariant of class LIST

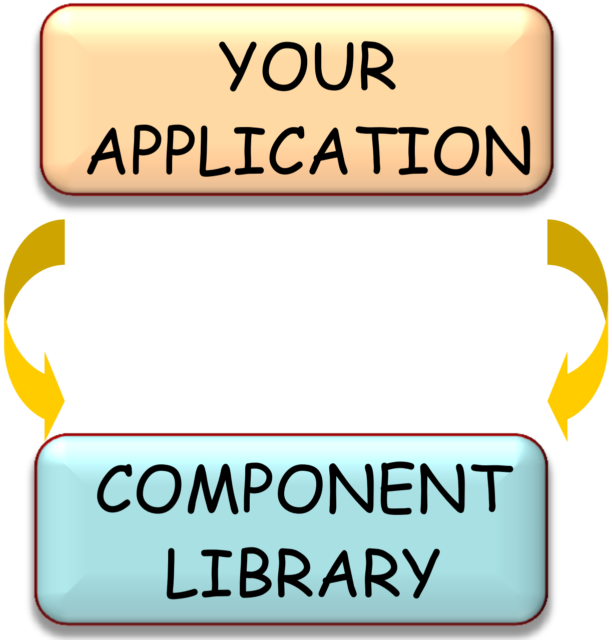


```
invariant

    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

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)
        require
            i >= 0
            i <= count + 1
```

Next step: automated testing”

B. Meyer et al., *Programs that test themselves*, IEEE Computer, Sept. 2009

What can be automated:

- Test suite execution
- Resilience (continue test process after failure)
- Regression testing
- Test case generation
- Test result verification (*oracles*)
- Test extraction from failures
- Test case minimization

Contracts for testing

Contracts provide the right basis:

- A fault is a discrepancy between intent and reality
- Contracts describe intent

A contract violation always signals a fault:

- Precondition: in client
- Postcondition or invariant: in routine (supplier)

In EiffelStudio: select compilation option for contract monitoring at level of class, cluster or system.

- 4 -

Contracts & analysis, methodological notes

Precondition design

The client must guarantee the precondition before the call

This does not necessarily mean testing for the precondition

Scheme 1 (testing):

```
if not my_stack.is_full then
        my_stack.put (some_element)
end
```

Scheme 2 (guaranteeing without testing):

```
my_stack.remove
...
my_stack.put (some_element)
```

Another example

```
sqrt (x, epsilon: REAL): REAL
                -- Square root of x, precision epsilon
        require
                x >= 0
                epsilon > 0
        do
                ...
        ensure
                abs (Result ^ 2 - x) <= 2 * epsilon * Result
        end
```

The contract

sqrt	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Provide non-negative value and precision that is not too small.	(From postcondition:) Get square root within requested precision.
Supplier	(Satisfy postcondition:) Produce square root within requested precision.	(From precondition:) Simpler processing thanks to assumptions on value and precision.

Not defensive programming!

It is never acceptable to have a routine of the form

```
sqrt (x, epsilon : REAL): REAL
            -- Square root of x, precision epsilon
      require
            x >= 0
            epsilon > 0

      do
            if x < 0 then
                  … Do something about it (?) …
            else
                  … Normal square root computation …
            end
      ensure
            abs (Result ^ 2 - x) <= 2 * epsilon * Result

      end
```

Not defensive programming

For every consistency condition that is required to perform a certain operation:

- Assign responsibility for the condition to one of the contract's two parties (supplier, client).
- Stick to this decision: do not duplicate responsibility.

Simplifies software and improves global reliability.

Interpreters

```
class BYTECODE_PROGRAM  feature
    verified : BOOLEAN
    trustful_execute (program: BYTECODE)
        require
                ok : verified
        do ... end

    distrustful_execute (program: BYTECODE)
        do
                verify
                if verified then trustful_execute ( program)
end
        end

    verify  do       ... end
end
```

How strong should a precondition be?

Two opposite styles:

- Tolerant: weak preconditions (including the weakest, *True*: no precondition).
- Demanding: strong preconditions, requiring the client to make sure all logically necessary conditions are satisfied before each call.

Partly a matter of taste.

But: demanding style leads to a better distribution of roles, provided the precondition is:

- Justifiable in terms of the specification only.
- Documented (through the short form).
- Reasonable!

The demanding style

```
sqrt (x, epsilon: REAL): REAL
        -- Square root of x, precision epsilon
        require
                x >= 0
                epsilon > 0
        do
                ...
        ensure
                abs (Result ^ 2 - x) <= 2 * epsilon * Result
        end
```

A tolerant style

```
sqrt (x, epsilon: REAL): REAL
        -- Square root of x, precision epsilon.
    require
        True
    do
        if x < 0 then
            ... Do something about it (?) ...
        else
            ... Normal square root computation ...
             computed := True
        end
    ensure
        computed implies abs (Result ^ 2 - x) <= 2 * epsilon * Result
    end
```

Contrasting styles

```
put (x: G)
            -- Push x on top of stack.
      require
            not is_full
      do
            ....
      end


tolerant_put (x: G)
      -- Push x if possible, otherwise set impossible to True.
do
      if not is_full then
            put (x)
      else
            impossible := True
      end
end
```

- 5 -

Contracts and inheritance

Contracts and inheritance

Issues: what happens, under inheritance, to

- Class invariants?
- Routine preconditions and postconditions?

Invariants

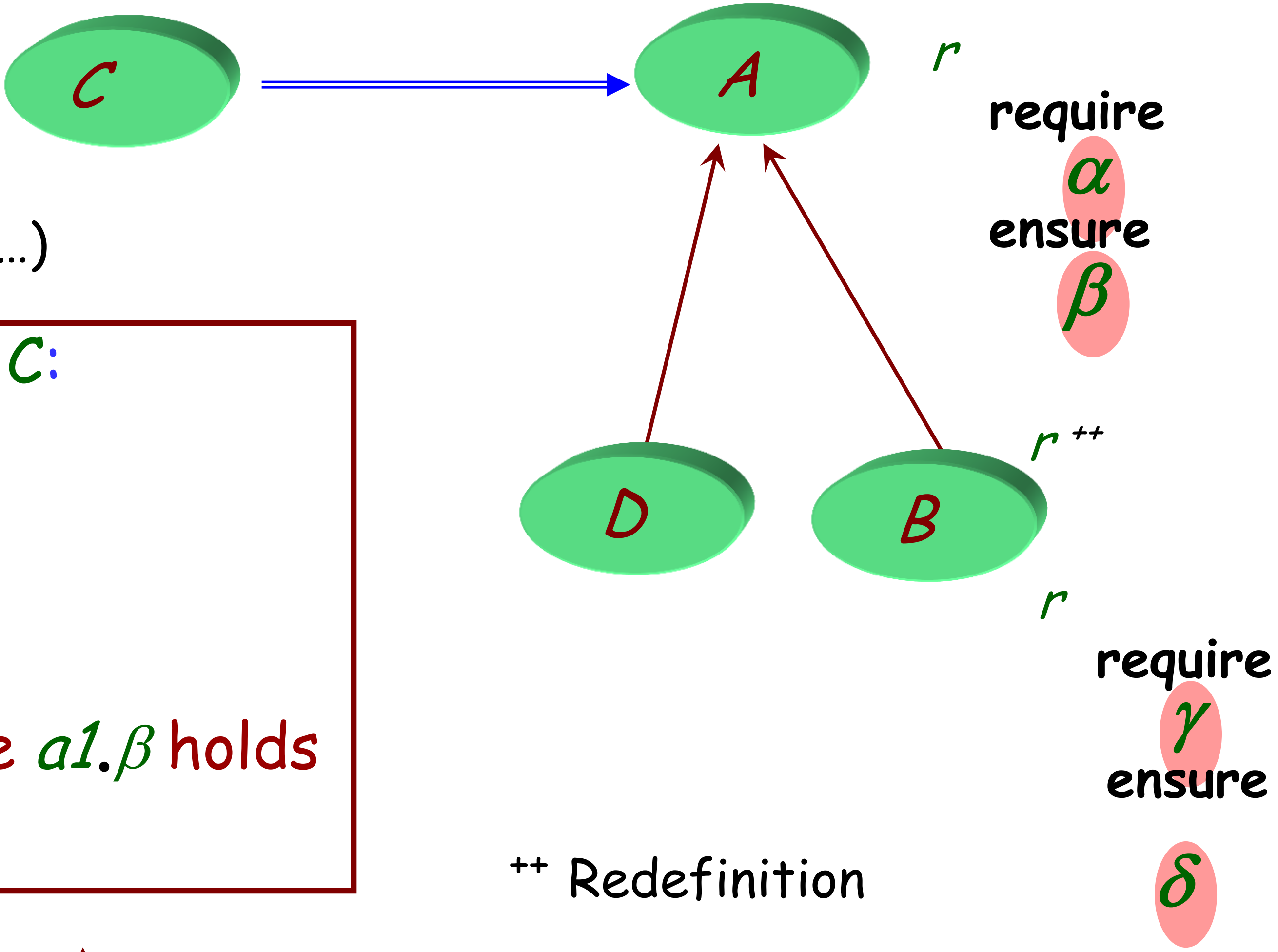
Invariant Inheritance rule:

- 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



a1: *A*

…

a1.*r* (…)

Correct call in *C*:

if *a1*.α **then**

    a1.*r* (...)

        -- Here *a1*.β holds

end

++ Redefinition

Client

Inheritance

Assertion redeclaration rule

When redeclaring a routine, we may only:

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

Assertion redeclaration rule

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*

- 6 -

Contracts & loops

Quiz: what does this function compute?

```
euclid (a, b: INTEGER): INTEGER
                -- Greatest common divisor of a and b
        require
                a > 0 ; b > 0
        local
                m, n : INTEGER
        do
                from
                                m := a ; n := b
                until
                                m = n
                loop
                                if m > n then
                                                m := m - n
                                else
                                                n := n - m
                                end
                end
                Result := m
        end
```

Quiz: what does this function compute?

```
euclid (a, b: INTEGER): INTEGER
        -- Greatest common divisor of a and b
    require
        a > 0 ; b > 0
    local
        m, n : INTEGER
    do
         from
                m := a ; n := b
        invariant
                -- "????????"
        variant
                ????????
        until
                m = n
        loop
                if m > n then
                        m := m - n
                else
                        n := n - m
                end
        end
        Result := m
    end
```

Quiz: what does this function compute?

```
euclid (a, b: INTEGER): INTEGER
        -- Greatest common divisor of a and b
    require
        a > 0 ; b > 0
    local
        m, n : INTEGER
    do
        from
                m := a ; n := b
        invariant
                -- gcd (m, n) = gcd (a, b)
        variant
                ????????
        until
                m = n
        loop
                if m > n then
                        m := m - n
                else
                        n := n - m
                end
        end
        Result := m
    end
```

Loop invariant

True after loop initialization

Preserved by loop body (i.e. if true before, will be true afterwards) when exit condition not true

```
from
        Init
until
        Exit
loop
        Body
end
```

Quiz: what does this function compute?

```
euclid (a, b: INTEGER): INTEGER
        -- Greatest common divisor of a and b
    require
        a > 0 ; b > 0
    local
        m, n : INTEGER
    do
        from
            m := a ; n := b
        invariant
            -- gcd (m, n) = gcd (a, b)
        variant
            ????????
        until
            m = n
        loop
            if m > n then
                m := m − n
            else
                n := n − m
            end
        end
        Result := m
    end
```

Loop variant

Integer expression that must:

- Be non-negative when after initialization (**from**)
- Decrease (i.e. by at least one), while remaining non-negative, for every iteration of the body (**loop**) executed with exit condition *not* satisfied

Quiz: what does this function compute?

```
euclid (a, b: INTEGER): INTEGER
        -- Greatest common divisor of a and b
    require
        a > 0 ; b > 0
    local
        m, n : INTEGER
    do
         from
                m := a ; n := b
        invariant
                -- gcd (m, n) = gcd (a, b)
        variant
                max (m, n)
        until
                m = n
        loop
                if m > n then
                        m := m - n
                else
                        n := n - m
                end
        end
        Result := m
    end
```

Invariants: loops as problem-solving strategy

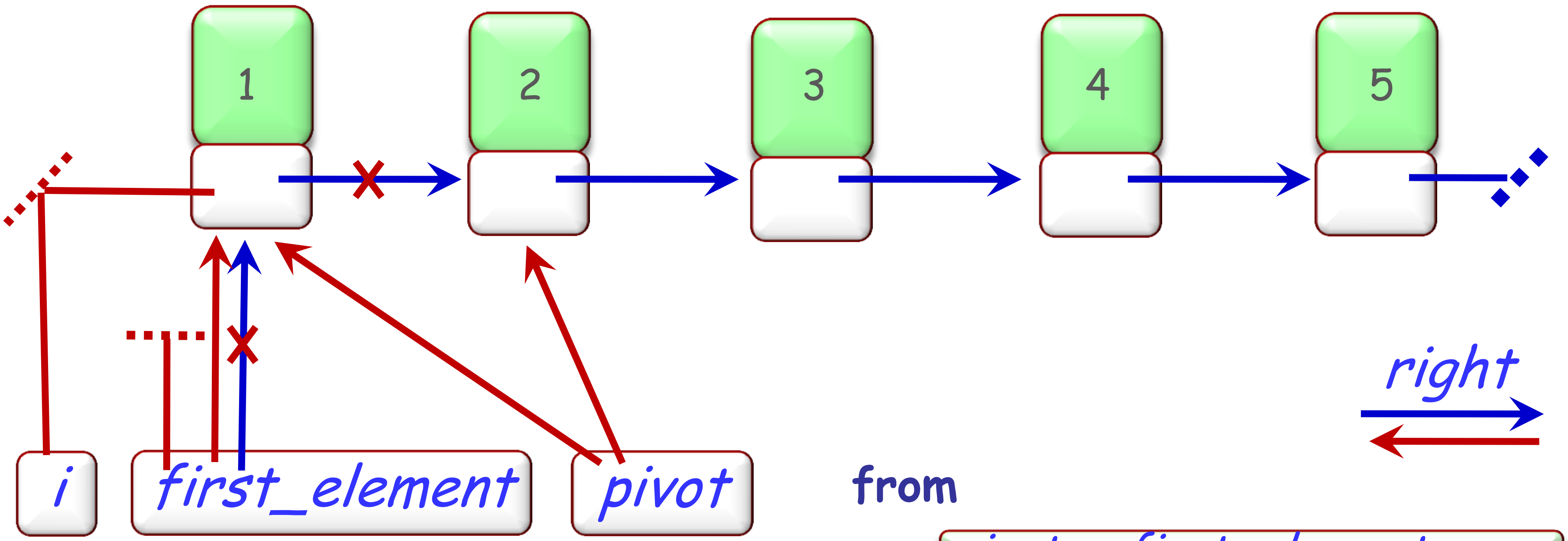
A loop invariant is a property that:

- Is easy to **establish initially**
  (even to cover a trivial part of the data)
- Is easy to **extend** to cover a bigger part
- If covering all data, gives the **desired result**!

Computing the maximum of a list

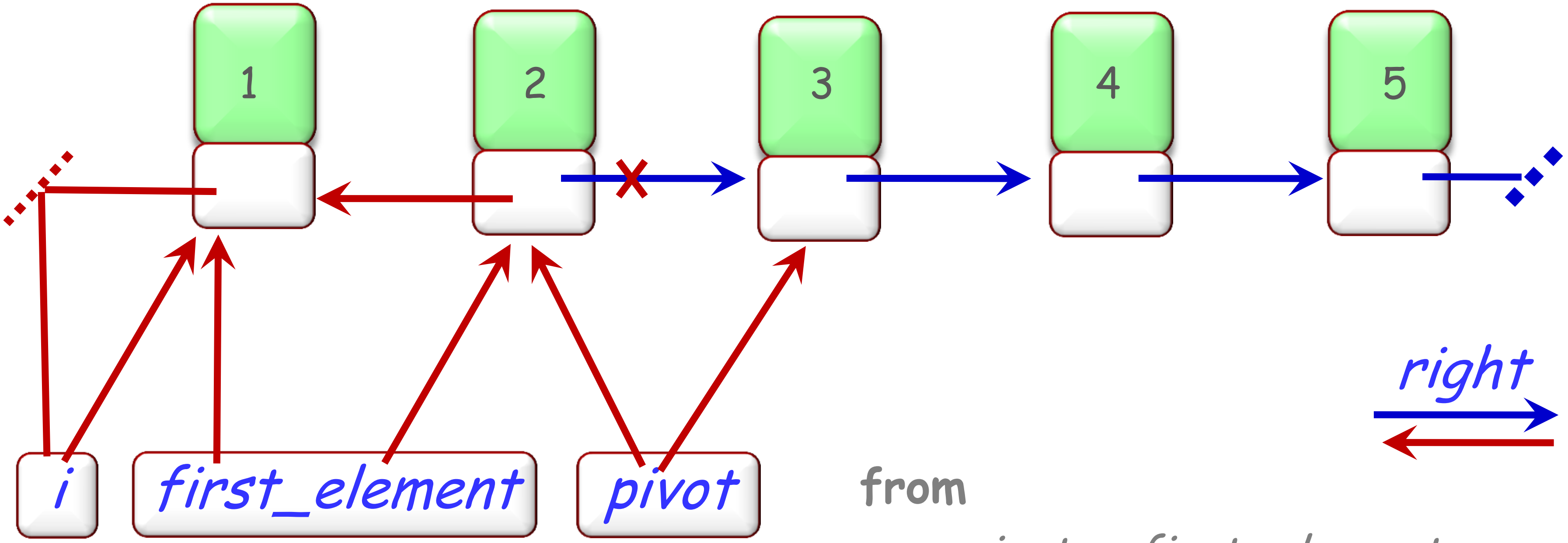
```
from
        ???
invariant
        ???
across structure as i loop
        Result := max (Result, i.item)
end
```

Reversing a list



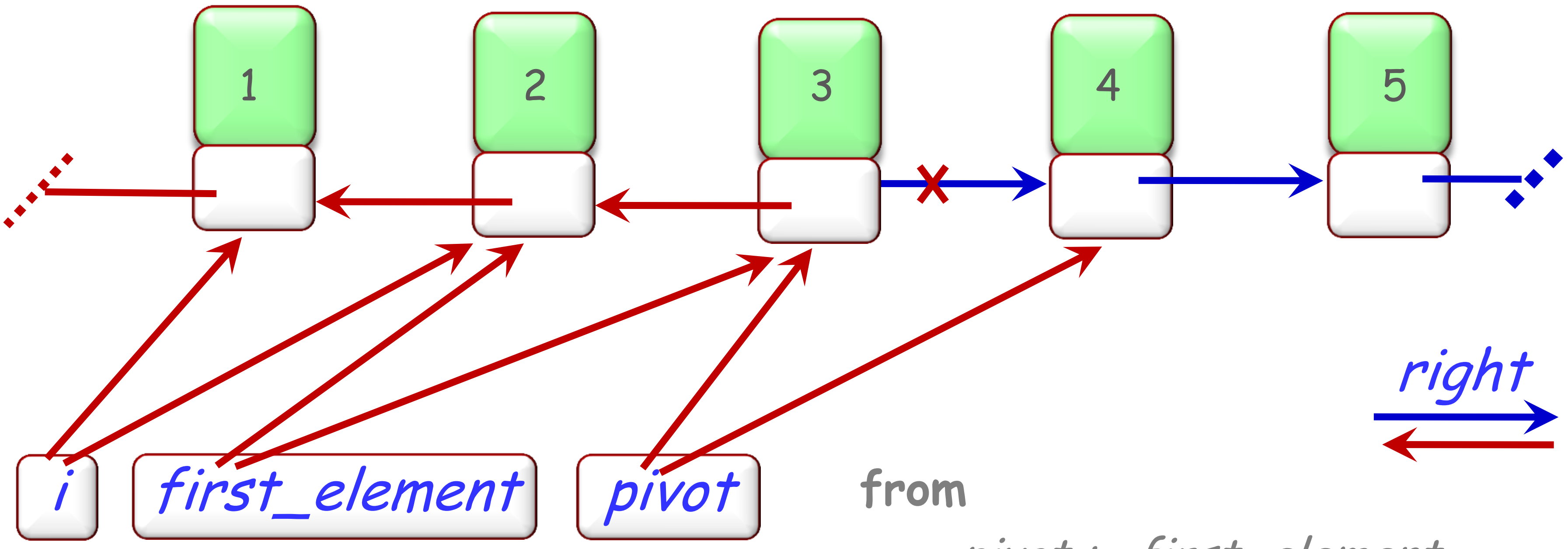
```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Reversing a list



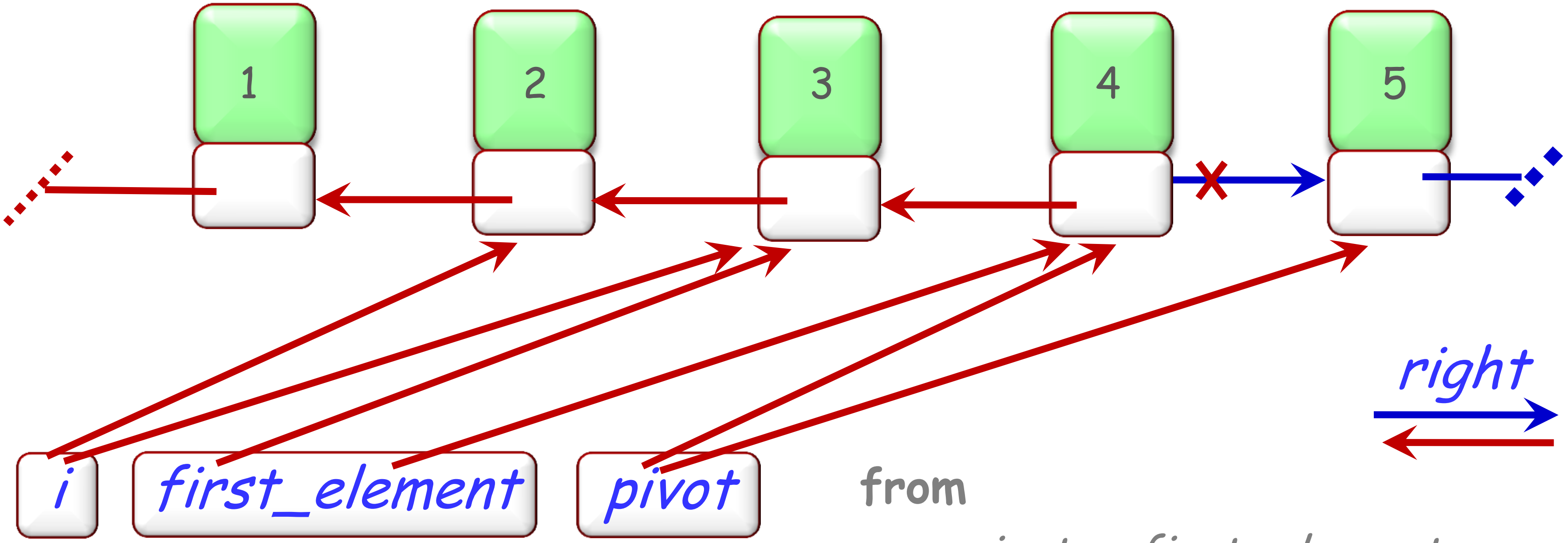
```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



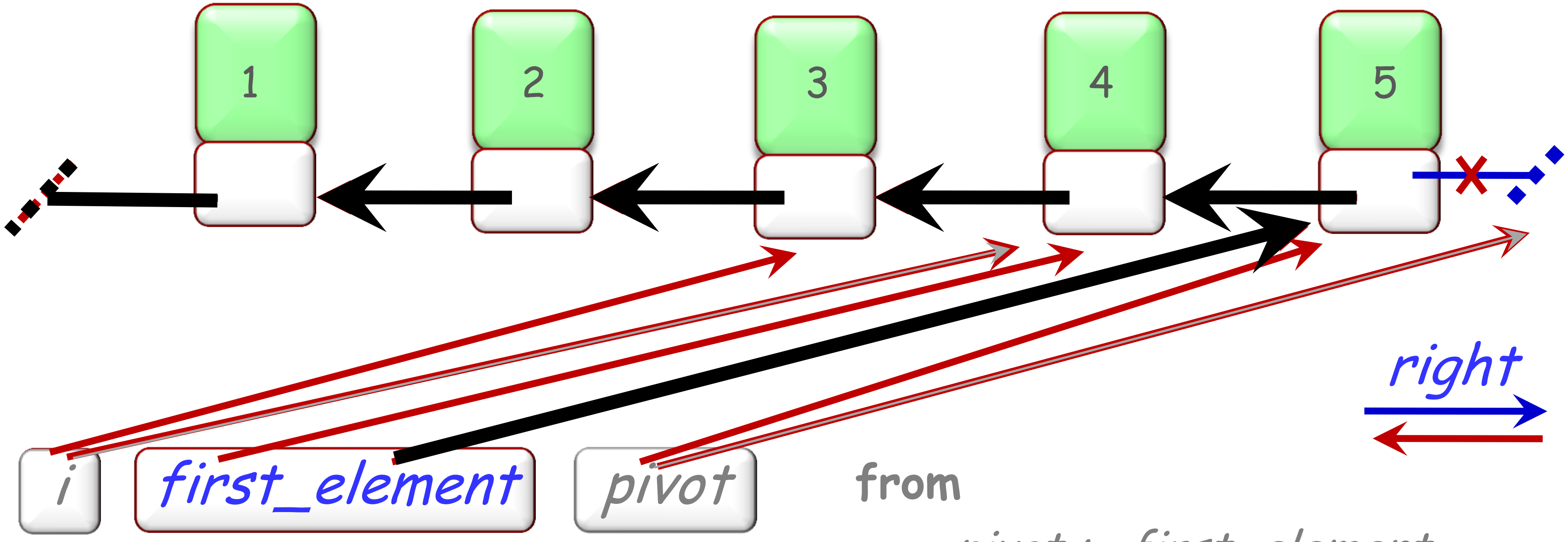
```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



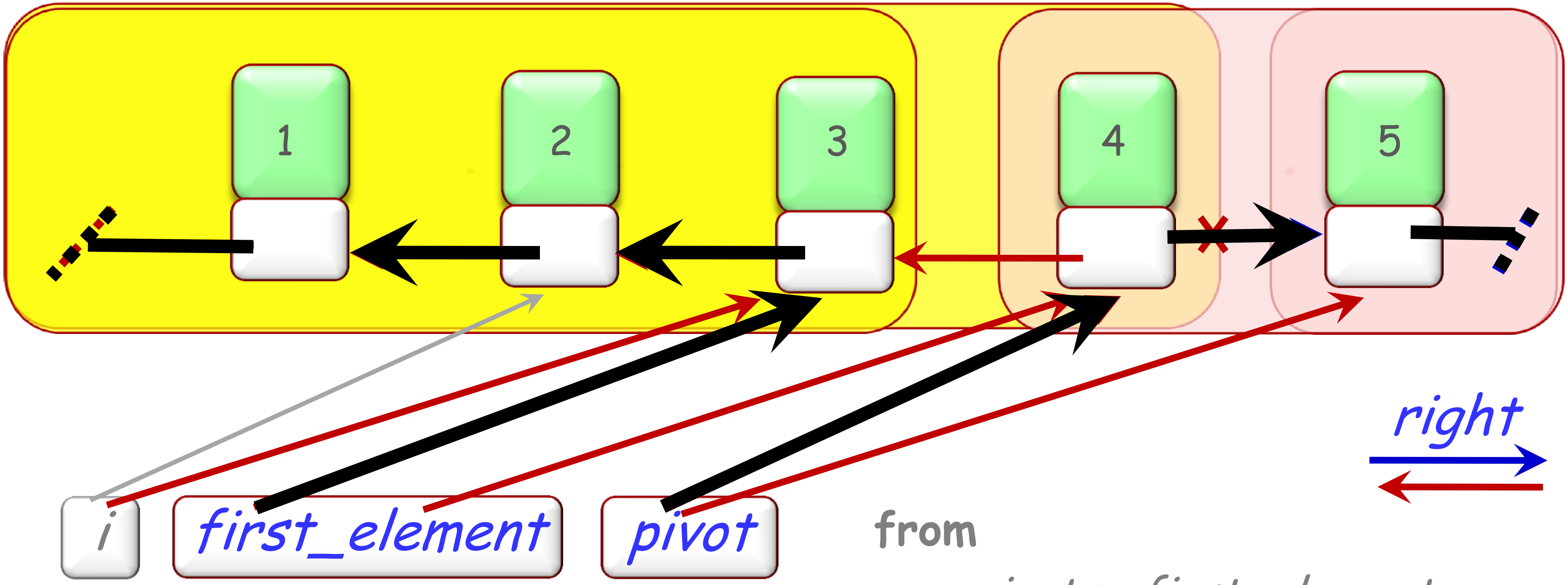
```
from
        pivot := first_element
        first_element := Void
until pivot = Void loop
        i := first_element
        first_element := pivot
        pivot := pivot.right
        first_element.put_right (i)
end
```

Reversing a list



```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Why does it work?



```
from
    pivot := first_element
    first_element := Void
until pivot = Void loop
    i := first_element
    first_element := pivot
    pivot := pivot.right
    first_element.put_right (i)
end
```

Invariant: from *first_element* following *right*, initial items in inverse order; from *pivot*, rest of items in original order

- 6 -

Handling abnormal cases

Abnormal case

An "abnormal case" is a case of applying a partial function outside of its domain

5 approaches:

- 1. A priori check
- 2. A posteriori check
- 3. Using agents
- 4. Return codes
- 5. Exception handling

Exception handling

Things not always happen in the ideal way!

Solution 1: Use standard control structures

```
if not end_of_file then
        read_token
        if token /= “class” then
                message (“File must start with class”)
        else
                read_token
                if not token.is_identifier then
                        message (“Invalid class name”)
                else
                        if token.name.is_taken then
                                message (“Class name in use”)
                        else
                                ...
```

Solution 1: a priori (check before)

```
if y.property then

        a.f (y)

else

        ...

end
```

```
f (x: T )
        require
                x.property
        do
                ...
        ensure
                Result.other_property
        end
```

Example: linear equation

Purpose: solve $A * x = b$, given matrix A and vector b
(the result x will be a vector)

```
if      A.regular      then
        x := A.solution (b)
else
        ...
end
```

Solution 1: a priori (check before)

```
if y.property then
        a.f (y)
else
        ...
end
```

```
f (x: T )
        require
                x.property
        do
                ...
        ensure
                Result.other_property
        end
```

Solution 2: a posteriori (try and check)

```
a.try_f (y)

if  it_worked then

        ... Continue normally ...
else

        ...

end
```

Solution 1:

```
if y.property then
        a.f (y)
else
        ...
end
```

```
f (x: T )
    require
        x.property
    do
        ...
    ensure
        Result.other_property
    end
```

Linear equation with solution 2

```
A.invert (b)

if A.is_inverted then

        x := A.solution

        ... Continue normally ...
else

        ...

end
```

Solution 1:

```
    if A.regular then
        x := A.solution (b)
    else
            ...
    end
```

Solution 3: using agents

Scheme 1:

```
action1
if ok1 then
        action2
        if ok2 then
                action3
                -- More processing,
                -- more nesting ...
        end
end
```

Scheme 2:

```
controlled_execute ([
  agent action1,
  agent action2 (...),
  agent action3 (...)
  ])
if glitch then
  warning (glitch_message)
end
```

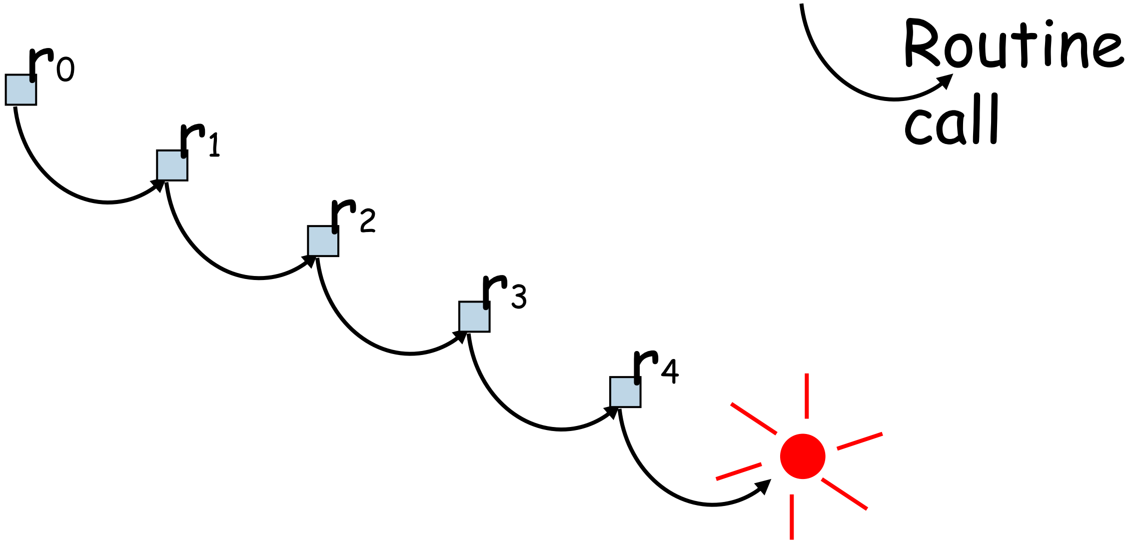
Solution 4: return codes

```
if (file_open (f)) {
          ... Continue with processing
          }
else
          {
          ...
          }
```

Solution 5: exceptions

In case of an abnormal situation:

- Interrupt execution
- Go up call chain
- If exception handler found, execute it
- Otherwise, program stops abnormally



What is an exception?

"An abnormal event"

Not a very precise definition

Informally: something that you don't want to happen…

Exception vocabulary

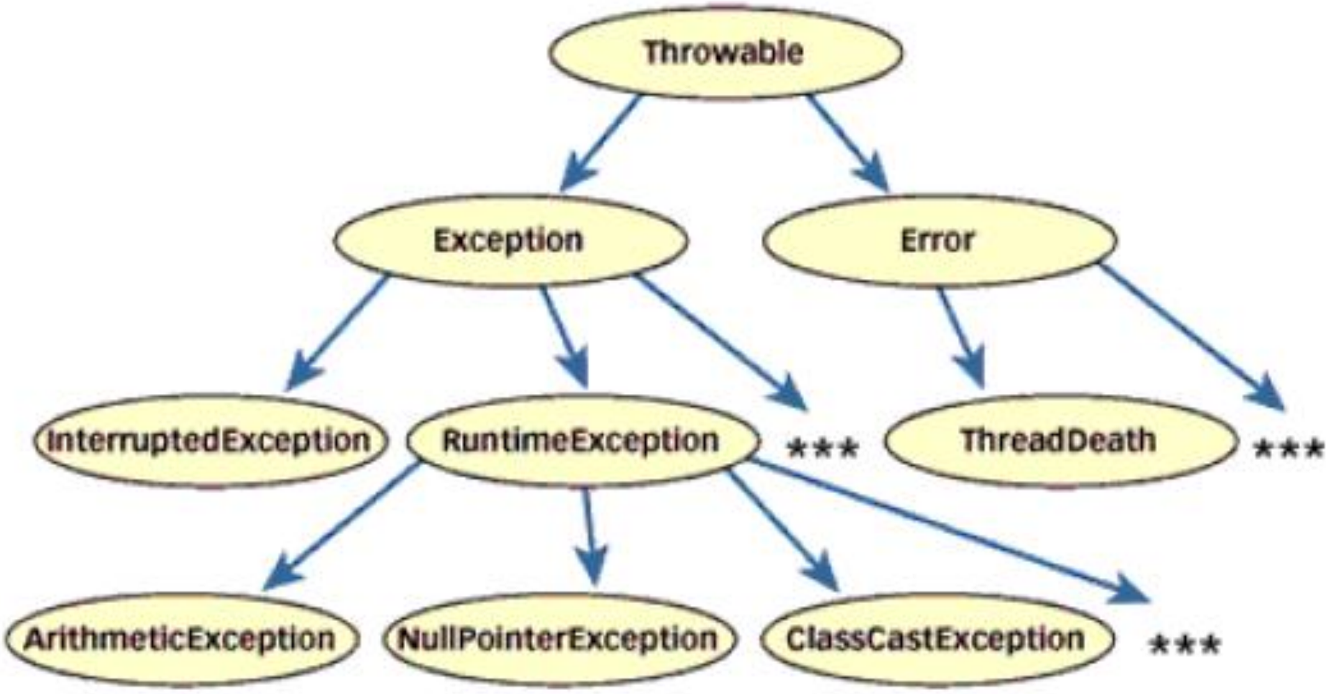
- "Raise", "trigger" or "throw" an exception
- "Handle" or "catch" an exception

C++/Java exception handling style

```
try {
        … Normal instructions, during which an
          exception may occur …
} catch (ET1 e) {
        … Handle exceptions of type ET1, details in e …
} catch (ET2 e) {
        … Handle exceptions of type ET2, details in e …
}… Possibly more cases…
finally {
        … Processing common to all cases, exception or not…
}
```

Java exceptions

Exceptions are objects, descendants of Throwable:



Java: raising an exception

Instruction:

```
throw  my_exception
```

The enclosing routine should be of the form

```
my_routine (…) throws my_exception {
        …
        if abnormal_condition
                throw  my_exception;
    }
```

How to use exceptions?

Two opposite styles:

- Exceptions as a control structure:
  Use an exception to handle all cases other than the most favorable ones

  (e.g. a key not found in a hash table triggers an exception)

- Exceptions as a technique of last resort

Exception handling

A formal basis:

- Introduce notion of contract
- The need for exceptions arises when a contract is broken by either of its parties (client, supplier)

Two concepts:

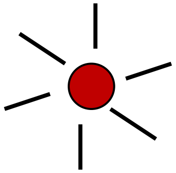
- Failure: a routine, or other operation, is unable to fulfill its contract.
- Exception: an undesirable event occurs during the execution of a routine — as a result of the failure of some operation called by the routine.

The original strategy

```
r (...)
        require
                ...
        do
                $op_1$
                $op_2$
                ...
                $op_i$
                ...
                $op_n$
        ensure
                ...
        end
```

Not going according to plan

```
r (...)
        require
                ...
        do
                op 1
                op 2
                ...
                op i
                ...
                opn
        ensure
                ...
        end
```



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

Causes of exceptions in O-O programming

Three major kinds:

- Operating system signal: arithmetic overflow, no more memory, interrupt ...
- Assertion violation (if contracts are being monitored)
- Void call (*x.f* with no object attached to *x*)

In Eiffel & Spec#, will go away

Handling exceptions properly

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"

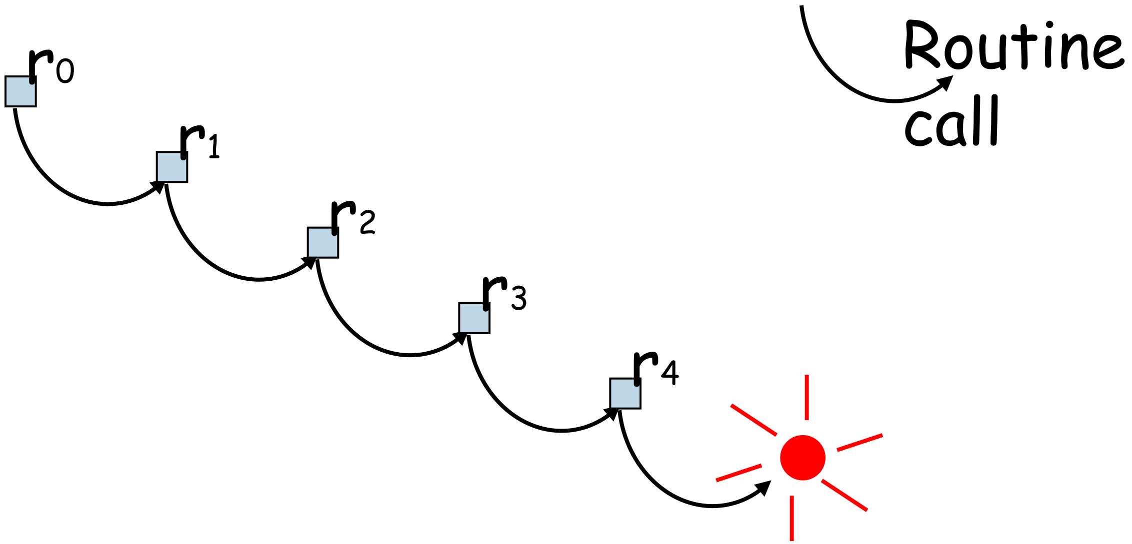
(Rare third case: false alarm)

How not to do it

(From an Ada textbook)

```
sqrt (x: REAL) return REAL is
        begin
                if x < 0.0 then
                        raise Negative;
                else
                        normal_square_root_computation;
                end
        exception
                when Negative =>
                        put ("Negative argument");
                        return;
                when others => ...
        end; -- sqrt
```

The call chain



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)

```
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
```

Another Ada textbook example

```
procedure attempt is begin
   <<Start>> -- Start is a label
   loop
      begin
         algorithm_1;
         exit; -- Alg. 1 success
      exception
         when others =>
            begin
               algorithm_2;
               exit; -- Alg. 2 success
            exception
               when others =>
                  goto Start;
               end
            end
         end
end main;
```

In Eiffel

```
attempt
  local
      even: BOOLEAN
  do
      if even then algorithm_2 else
          algorithm_1
      end
  rescue
      even := not even; retry
end
```

Dealing with arithmetic overflow

```
quasi_inverse (x: REAL): REAL
            -- 1/x if possible, otherwise 0
      local
            division_tried: BOOLEAN
      do
            if not division_tried then
                  Result := 1/x
            end
      rescue
            division_tried := True
            retry
      end
```

If no exception clause (1)

Absence of a rescue clause is equivalent, in first approximation, to an empty rescue clause:

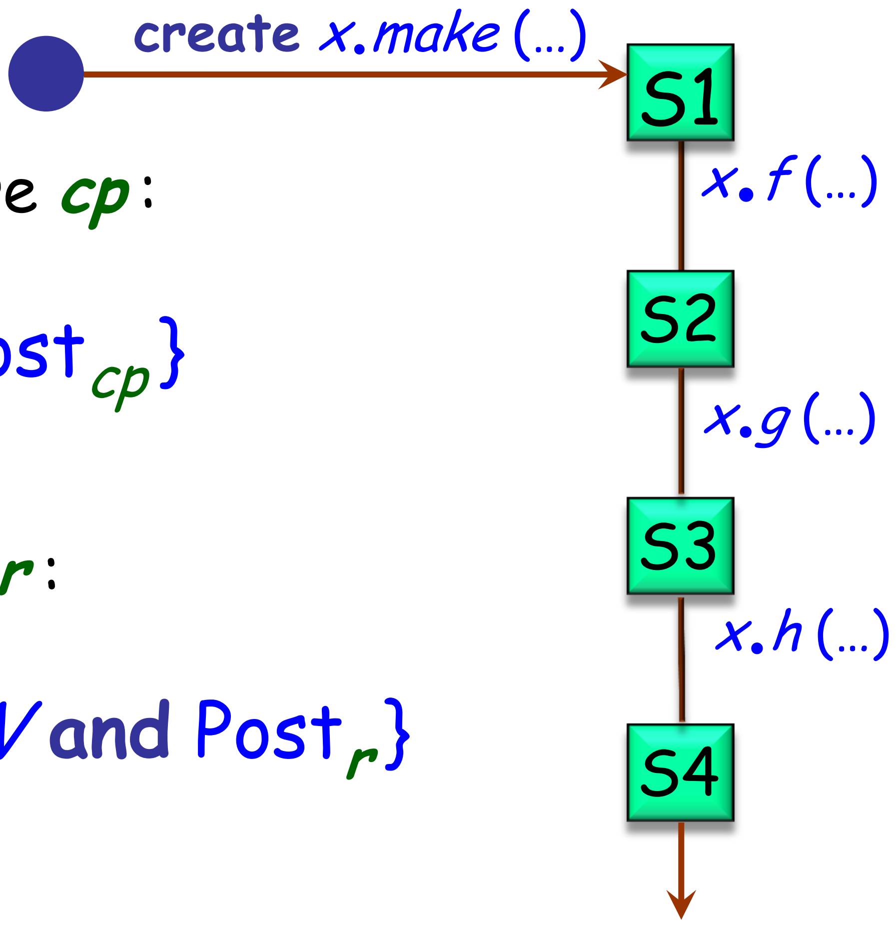
```
f (...)
        do
                ...
        end
```

is an abbreviation for

```
f (...)
        do
                ...
        rescue
                -- Nothing here
        end
```

(This is a provisional rule; see next.)

The correctness of a class



For every creation procedure *cp* :

$\{Pre_{cp}\}$ **do**$_{cp}$ $\{INV \text{ and } Post_{cp}\}$

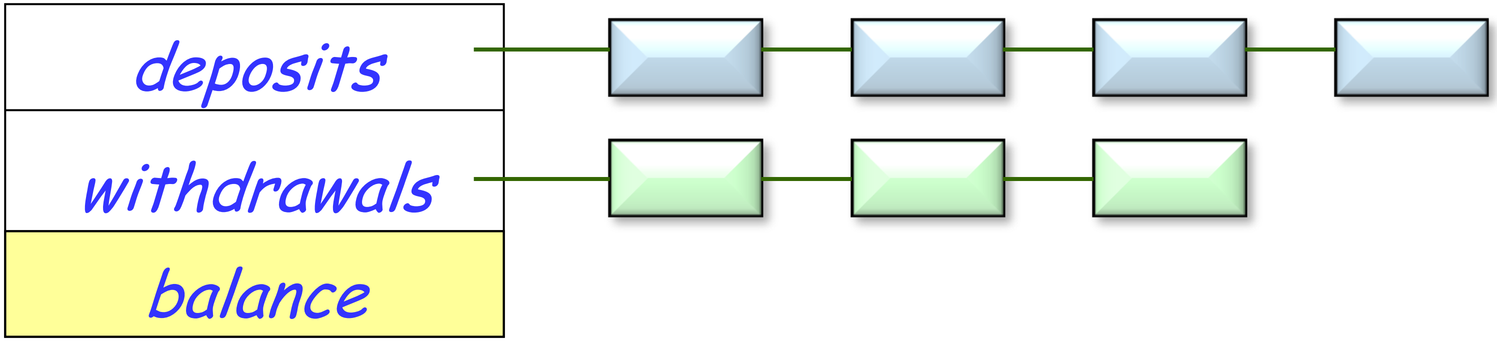
For every exported routine *r* :

$\{INV \text{ and } Pre_r\}$ **do**$_r$ $\{INV \text{ and } Post_r\}$

Bank accounts

balance := *deposits*.*total* - *withdrawals*.*total*

(A2)



Exception correctness

For the normal body:

$$\{INV \textbf{ and } Pre_r\}\ \textbf{do}_r\ \{INV \textbf{ and } Post_r\}$$

For the exception clause:

$$\{???\}\ \textbf{rescue}_r\ \{???\}$$

Exception correctness

For the normal body:

$$\{INV \text{ and } Pre_r\} \ \mathbf{do}_r \ \{INV \text{ and } Post_r\}$$

For the exception clause:

$$\{\mathbf{True}\} \ \mathbf{rescue}_r \ \{INV\}$$

If no exception clause (2)

Absence of a rescue clause is equivalent to a default rescue clause:

```
f (...)
        do
              ...
        end
```

is an abbreviation for

```
f (...)
        do
              ...
        rescue
              default_rescue
        end
```

The task of *default_rescue* is to restore the invariant.

For finer-grain exception handling

Every exception has a type, a descendant of the library class *EXCEPTION*

Query *last_exception* gives an object representing the last exception that occurred

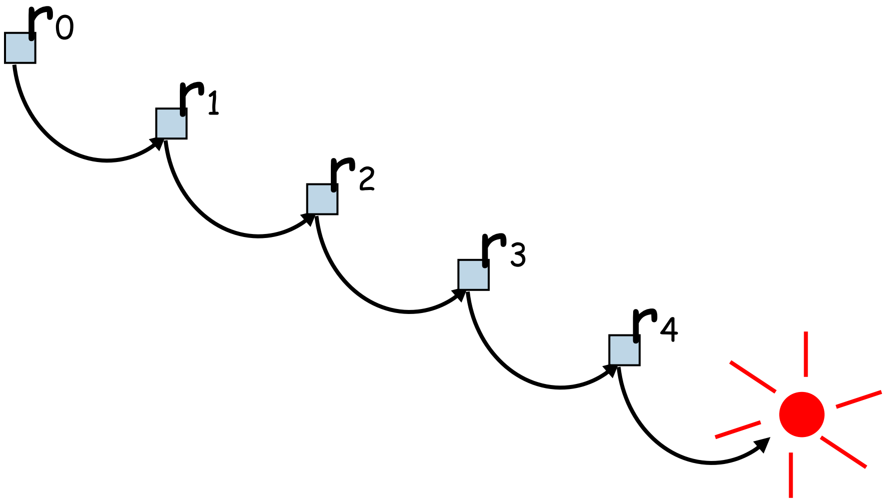
Some features of class *EXCEPTION*:

- *name*
- *is_assertion_violation*, etc.
- *raise*

Another challenge today

Exceptions in a concurrent world

What if the call chain
is no longer available?



Exception handling: summary and conclusion

Exceptions as a control structure (internally triggered):

- Benefits are dubious at best

- An exception mechanism is needed for unexpected external events

- Need precise methodology; must define what is “normal” and “abnormal”. Contracts provide that basis.

- Next challenge is concurrency & distribution

- 8 -

Design by Contract in various languages

What we do with contracts

Write better software

Analyze

Design

Reuse

Implement

Use inheritance properly

Avoid bugs

Document software automatically

Help project managers do their job

(with run-time monitoring)

Perform systematic testing

Guide the debugging process

Emulating Design by Contract mechanisms

Basic step (programmer discipline):

- Add preconditions and postconditions
- Use switch to turn monitoring on or off
- Help for analysis, methodology, debugging, but
  - No documentation help
  - No class invariants
  - No connection with O-O structure
  - No inherited assertions
  - No connection with exception handling

Other techniques:

- Macros (C, C++)
- Language extensions, e.g. preprocessor

The macro approach

GNU Nana: improved support for contracts and logging in C and C++.

Set of C++ macros and commands for gdb debugger. Replaces assert.h.

Support for quantifiers (Forall, Exists, Exists1) corresponding to iterations on the STL (C++ Standard Template Library).

Support for time-related contracts ("Function must execute in less than 1000 cycles").

Gnu Nana example

```
void intsqrt(int &r) { /* r’ = floor(sqrt(r)) */
DS($r = r); /* save r away into $r for later use under gdb(1) */
          DS($start = $cycles); /* real time constraints */
          ...; /* code which changes r */
          DI($cycles – $start < 1000);
                    /* code must take less than 1000 cycles */
          DI(((r * r) <= $r) && ($r < (r + 1) * (r + 1)));
                    /* use $r in postcondition */
          }
```

Design by Contract in Java

OAK 0.5 (pre-Java) contained an assertion mechanism, which was removed due to “lack of time”.

Several different proposals. Gosling (May 1999, http://www.javaworld.com/javaworld/javaone99/j1-99-gosling.html):

“ *The number one thing people have been asking for is an assertion mechanism. Of course, that [request] is all over the map: There are people who just want a compile-time switch. There are people who ... want something that's more analyzable. Then there are people who want a full-blown Eiffel kind of thing. We're probably going to start up a study group on the Java platform community process."*

Java Modeling Language (JML)

Contract-equipped extension for Java

Assertions are in the form of JavaDoc comments

Rich tool suite for tests and proofs

JML example (1)

```
public class BankingExample {
    public static final int MAX_BALANCE = 1000;
    private /*@ spec_public @*/ int balance;
    private /*@ spec_public @*/ boolean isLocked = false;
    //@ public invariant balance >= 0 && balance <= MAX_BALANCE;
    //@ assignable balance;
    //@ ensures balance == 0;
    public BankingExample() { balance = 0; }

    //@ requires 0 < amount && amount + balance < MAX_BALANCE;
    //@ assignable balance;
    //@ ensures balance == \old(balance + amount);
    public void credit(int amount) { balance += amount; }
```

JML example (2)

```
 //@ requires 0 < amount && amount <= balance;
//@ assignable balance;
//@ ensures balance == \old(balance) - amount;
public void debit(int amount) { balance -= amount; }

//@ ensures isLocked == true;
public void lockAccount() { isLocked = true; }

//@   requires !isLocked;
//@   ensures \result == balance;
//@ also
//@   requires isLocked;
//@   signals_only BankingException;
public /*@ pure @*/ int getBalance() throws BankingException {
    if (!isLocked) { return balance; }
    else { throw new BankingException(); }
}
```

Object Constraint Language

Contract extension to UML

Includes support for:

- Invariants, preconditions, postconditions
- Guards (not further specified).
- Predefined types and collection types
- Associations
- Collection operations: ForAll, Exists, Iterate

OCL example

Postconditions:

```
        post: result = collection->iterate
                (elem; acc : Integer = 0 | acc + 1)
        post: result = collection->iterate
                ( elem; acc : Integer = 0 |
                if elem = object then acc + 1 else acc endif)
post: T.allInstances->forAll
                (elem | result->includes(elem) = set->
                        includes(elem) and set2->includes(elem))
```

Spec#

Contract-equipped version of C# language

Originally developed at Microsoft Research

Includes non-null types

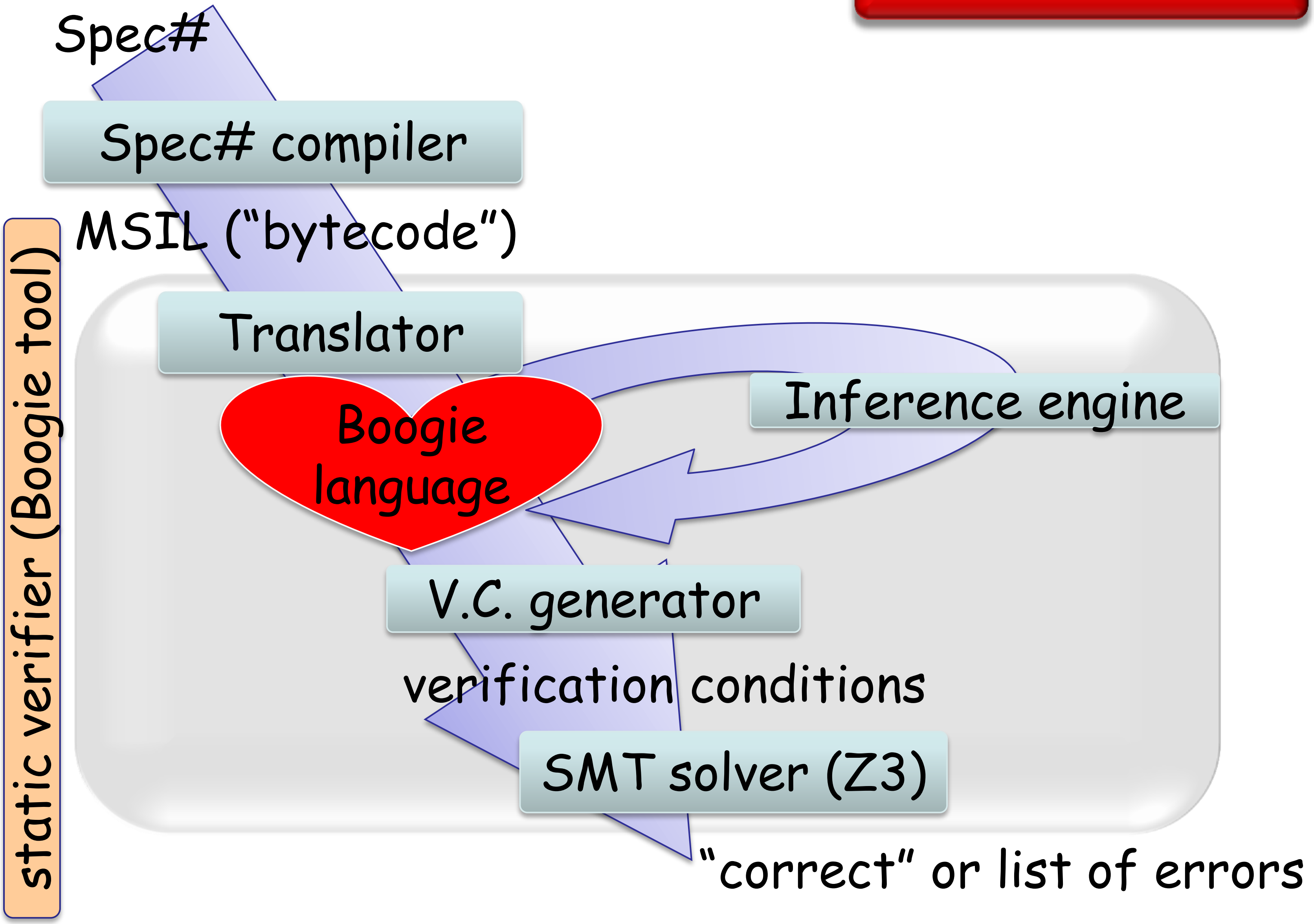
Spec# contract example

Source: Rustan Leino

```
static int min (int x, int y)
  requires 0 <= x && 0 <= y ;
  ensures  x < y ? result == x: result == y;
{
            int m;
            if (x < y)
                  m = x;
            else
                  m = y;
            return m;
}
```

The Spec# verifier

Source: Rustan Leino



Spec#

Spec# compiler

MSIL ("bytecode")

Translator

Boogie language

Inference engine

V.C. generator

verification conditions

SMT solver (Z3)

"correct" or list of errors

static verifier (Boogie tool)

Code contracts

Introduced in 2009 to provide a "*language-agnostic way to express coding assumptions in .NET programs*" (Microsoft)

Set of static library methods for writing preconditions, postconditions, and "*object invariants*", with tools:

- **ccrewrite** to generate run-time checking
- **cccheck**: static checker
- **ccdoc**: for documentation

Applied to large part of mscore library

- 9 -

New developments

The next steps

Pushing some properties to the type system:

- Void safety

More expressive specifications

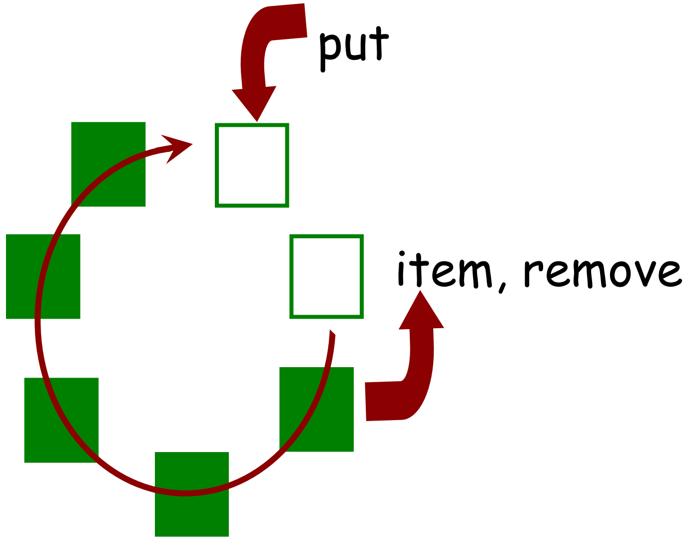
Concurrency

Proofs

Concurrency

SCOOP mechanism:

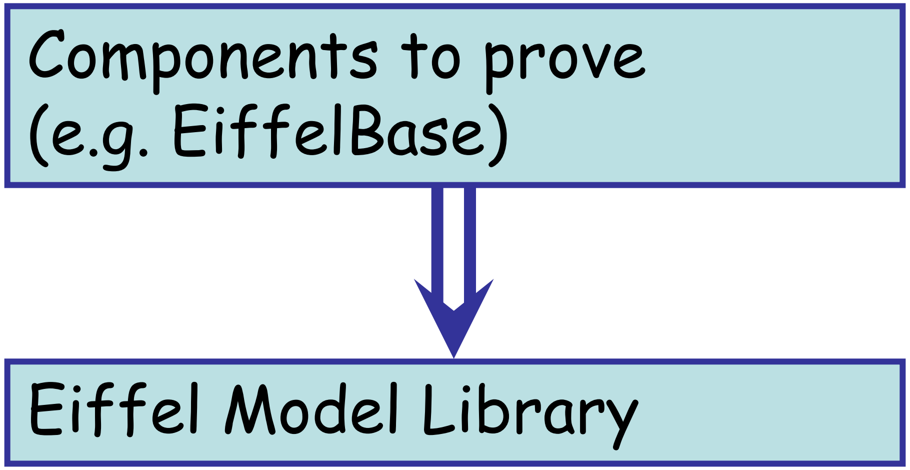
- General object-oriented notation for concurrent programs
- Based on reinterpretation of contracts: preconditions become wait conditions



```
put (b: QUEUE BUFFER [G ] ; v: G )
        -- Store v into b.
    require
        not b.is_full
    do
        ...
    ensure
        not b.is_empty
    end
```

```
my_queue : QUEUE BUFFER [T ]
...
if not my_queue.is_full then


      put (my_queue, t )
end
```

Increasing expressive power



Components to prove
(e.g. EiffelBase)

Eiffel Model Library

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)

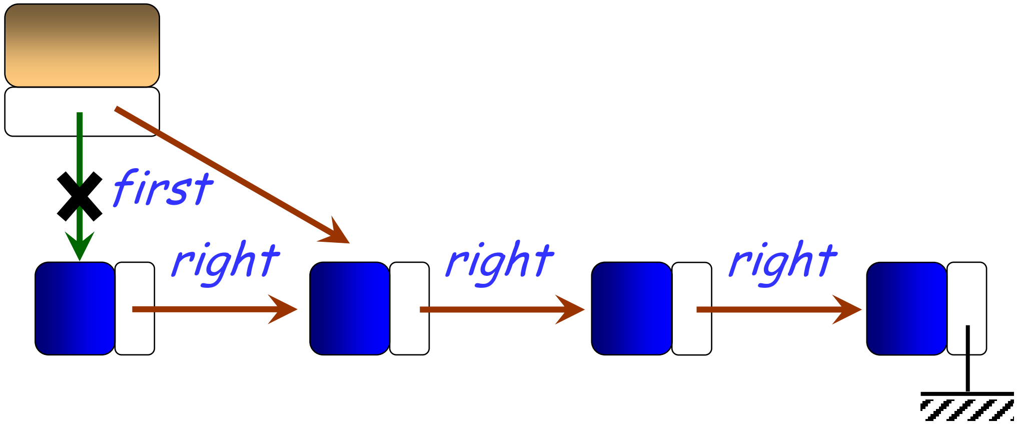
Specified with contracts (unproven) reflecting mathematical properties

Expressed entirely in Eiffel

Example MML class

```
class SEQUENCE [G] feature

            count : NATURAL
                        -- Number of items
            last : G
                        -- Last item

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

                        Result.last = x

                        Result.sub (1, count ) ~ Current

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

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
```

Principles

Very simple mathematics only

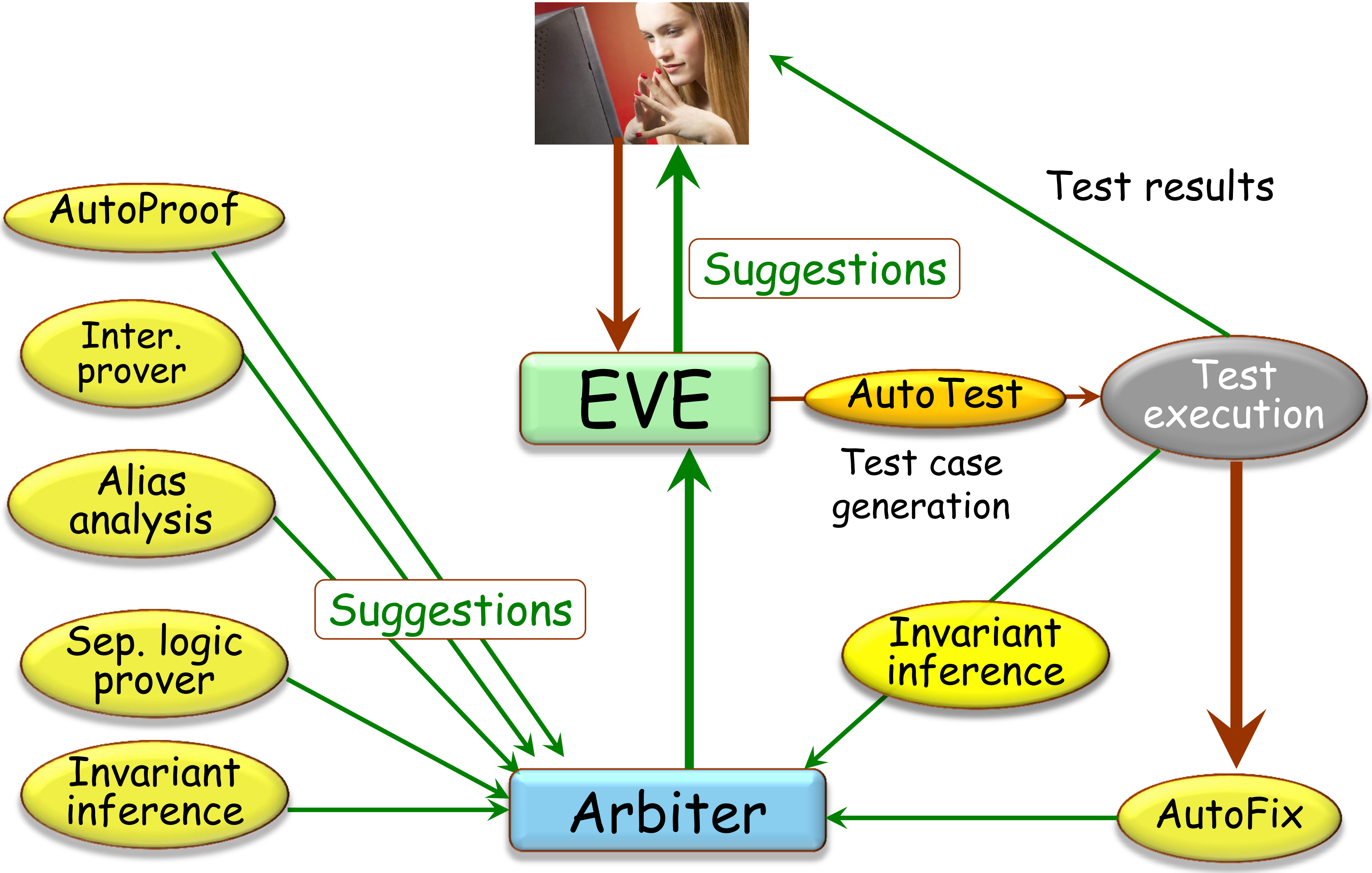
- Logic
- Set theory

EiffelBase2

Nadia Polikarpova

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



- 10 -

Conclusion

Design by Contract: technical benefits

More focused process: writing to spec

Sound basis for reuse

Exception handling guided by precise definition of “normal” and “abnormal” cases

Interface documentation automatically generated, up-to-date, can be trusted

Faults occur close to cause, found faster & more easily

Guide for black-box test case generation.

Design by Contract: managerial benefits

Library users can trust documentation

They can benefit from preconditions to validate their own software

Test manager can benefit from more accurate estimate of test effort

Black-box specification for free

Designers who leave bequeath not only code but intent

Common vocabulary between all actors of the process: developers, managers, potentially customers

Component-based development possible on a solid basis

Tom de Marco (IEEE Computer, Feb 1997)

"*I believe that the use of Eiffel-like module contracts is the most important non-practice in software today*"