Contracts, concurrency and SCOOP for Web Services

Bertrand Meyer

Professor of software engineering, ETH Zurich
Founder and chief architect, Eiffel Software

Trento, December 2003

The traditional programming world

- Sequential
- Used to be messy
- Still hard but:
  - Structured programming
  - Well-understood basic structures
  - Data abstraction & object technology
  - Design by Contract
  - Genericity, multiple inheritance
  - Architectural techniques
  - Much higher level than 20, 15, 10 years ago
- Switch from operational techniques to logical deduction (e.g. invariants) allows static reasoning

Design by Contract™

- Get things right in the first place
- Automatic documentation
- Self-debugging, self-testing code
- Proper handling of abnormal cases (exceptions, failures)
- Get inheritance right
- Give managers the right control tools

Contracts in Eiffel

```
store (buf: BUFFER [INTEGER]; value: INTEGER) is
        -- Store value into buffer.
    require
        not buf.is_full
        value > 0
    do
        buf.put (value)
    ensure
        not buf.is_empty
    end
...
```

Precondition: client obligation

Postcondition: supplier obligation

The traditional programming world

- Sequential
- Used to be messy
- Still hard but:
  - Structured programming
  - Well-understood basic structures
  - Data abstraction & object technology
  - Design by Contract
  - Genericity, multiple inheritance
  - Architectural techniques
  - Much higher level than 20, 15, 10 years ago
- Switch from operational techniques to logical deduction (e.g. invariants) allows static reasoning

The new world

- Everything has to be concurrent, distributed, multithreaded, pervasive, wearable, web-enabled...
- Used to be messy
- Still messy
  - Examples: threading models in most popular approaches
  - Development level: ca. 1968
  - Only understandable through operational reasoning

Concurrency & distribution

- Everyone wants to do it
- Many are doing it
- Those who are doing it are not doing it very well

Web services

- Generalization of client-server paradigm taking advantage of the ubiquity of the World-Wide Web

> **Definition**
>
> A Web Service is a service made available by a program to other programs through the World-Wide Web

- Technologies: HTTP, XML, SOAP, WSDL

Web service technologies

- HTTP (HyperText Transfer Protocol): the Web server protocol
- WSDL (Web Services Description Language): provide description of services offered
- UDDI: help search for services
- SOAP (Simple Object Access Protocol): encode service requests and results
- XML (eXceedingly Marketed Language): common format for all exchanges

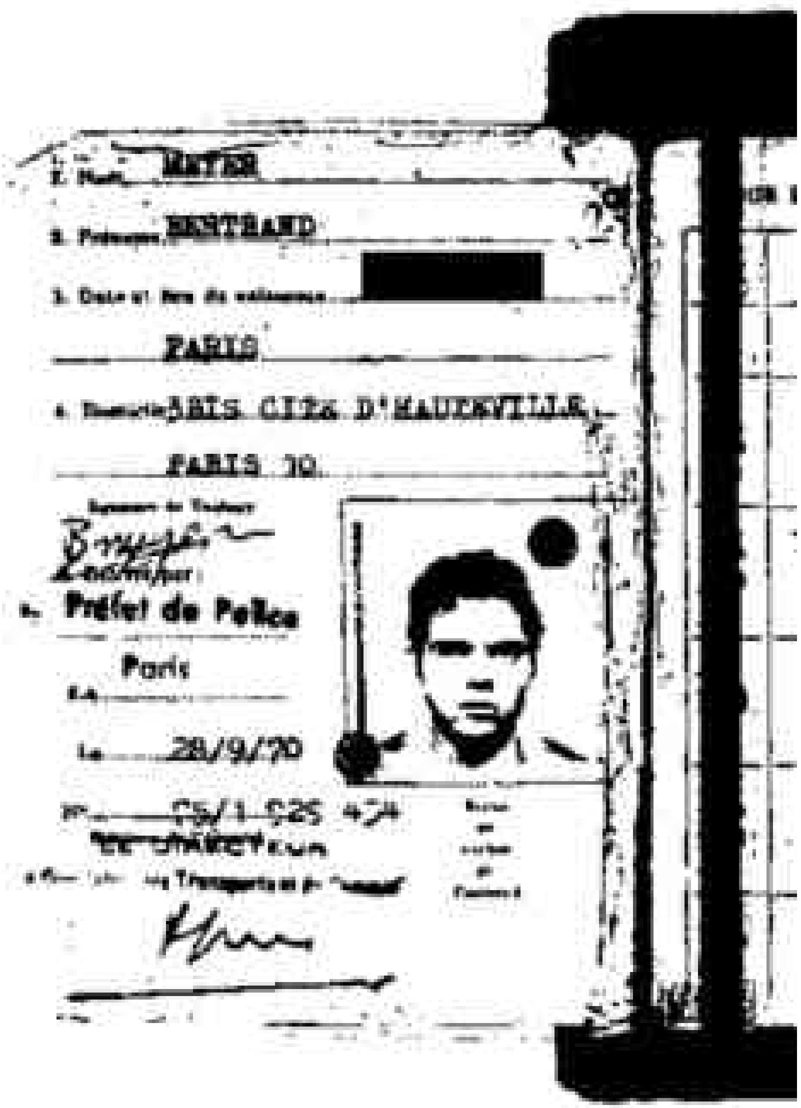
Some Web service issues

- Programming Web services
- Specifying the effect of a Web service
- Guaranteeing quality

The documentation problem

- How to guarantee that documentation is faithful to the software?
- How to guarantee that it *remains* faithful?
- How to get it in the first place?

The French Driver's License issue



The trouble with IDL

- Has to be written and maintained separately
- Better approach: Eiffel's contract form; .NET metadata
- Will WSDL reintroduce the problem?

The new world

- Everything has to be concurrent, distributed, multithreaded, pervasive, wearable, web-enabled...
- Used to be messy
- Still messy
  - Examples: threading models in most popular approaches
  - Development level: ca. 1968
  - Only understandable through operational reasoning

Impedance mismatch

- O-O: high-level abstraction mechanisms

- Concurrency: semaphores, locks, suspend, manual exclusion, sharing…

SCOOP

- Simple Concurrent Object-Oriented Programming
- First iteration 1990
- CACM, 1993
- Object-Oriented Software Construction, 2nd edition, 1997
- Prototype implementation at Eiffel Software, 1995
- Prototypes by others
- No being done for good at ETH, Hasler foundation funding, also ETH and Microsoft ROTOR project

Why O-O?

- Structuring concept: the class
  - Module-type fusion
  - Information hiding
  - Multiple inheritance
  - Genericity
  - Polymorphism and dynamic binding
  - Contracts

Computation concept: the object

- Modeling power
  - Dynamic allocation
  - Automatic memory management

x.r (a)

O-O and concurrency

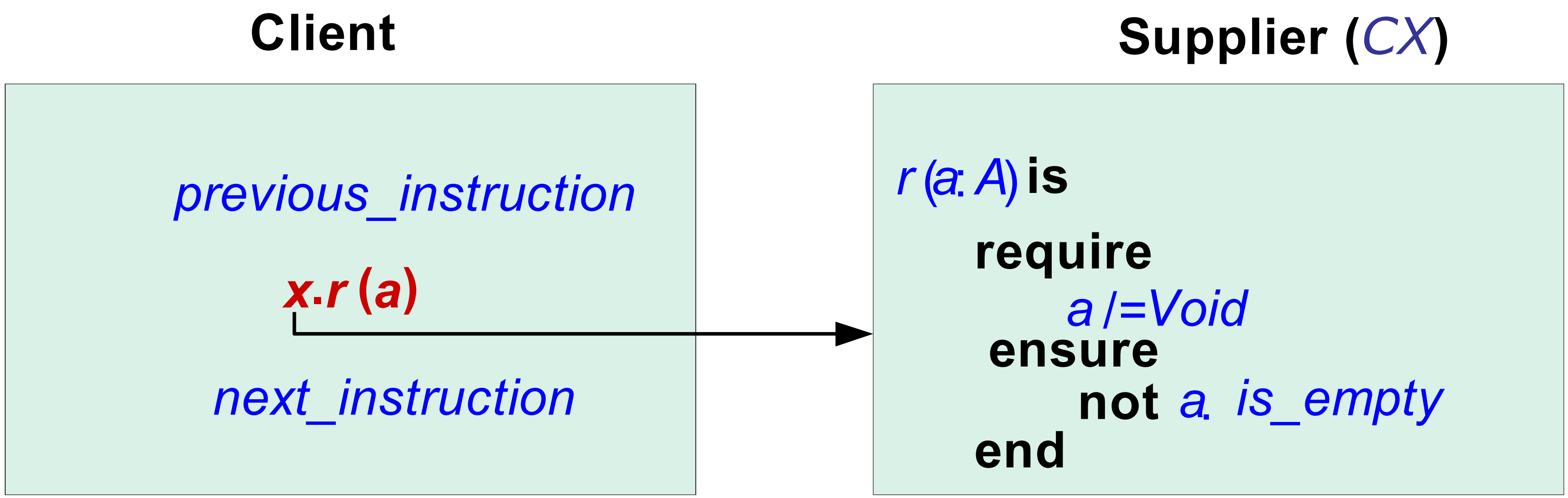
- "Objects are naturally concurrent" (Milner)
- Many attempts
- "Active objects"
- "Inheritance anomaly"

- No mechanism widely accepted
- In practice, low-level mechanisms on top of O-O language

Feature call

x: *CX*

x.r (a)



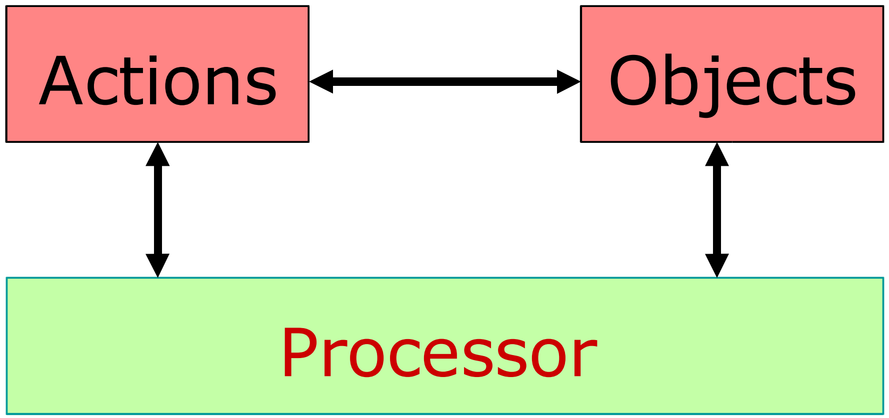
Processor

Object-oriented computation

To perform a computation is

- to apply certain actions
- to certain objects
- using certain processors

x.r (a)



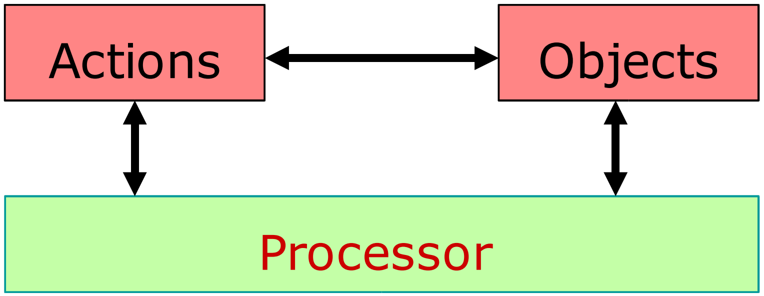
What makes an application concurrent?

Processor:

Thread of control supporting sequential execution of instructions on one or more objects

Can be implemented as:

- Computer CPU
- Process
- Thread
- AppDomain (.NET) …



Will be mapped to computational resources

Handling rule

All calls on an object
are executed by the processor's handler

Reasoning about objects

$$\frac{\{Pre_r \ \textbf{and} \ INV\} \quad body_r \quad \{Post_r \ \textbf{and} \ INV\}}{\{Pre_r'\} \ x.r\ (a) \quad \{Post_r'\}}$$

Reasoning about objects

Only *n* proofs if *n* exported routines!

$$\frac{\{Pre_r \textbf{ and } INV\} \quad body_r \quad \{Post_r \textbf{ and } INV\}}{\{Pre_r'\} \; x.r\,(a) \quad \{Post_r'\}}$$

In a concurrent context

Only *n* proofs if *n* exported routines?

$$\frac{\{Pre_r \ \textbf{and}\ INV\} \quad body_r \quad \{Post_r\ \textbf{and}\ INV\}}{\{Pre_r'\}\ x.r\ (a) \quad \{Post_r'\}}$$

Mutual exclusion rule

At most one feature may execute
on any one object at any one time

Feature call: sequential

x.r (a)

x: *CX*

Client

previous_instruction

x.r (a)

next_instruction

Supplier (*CX*)

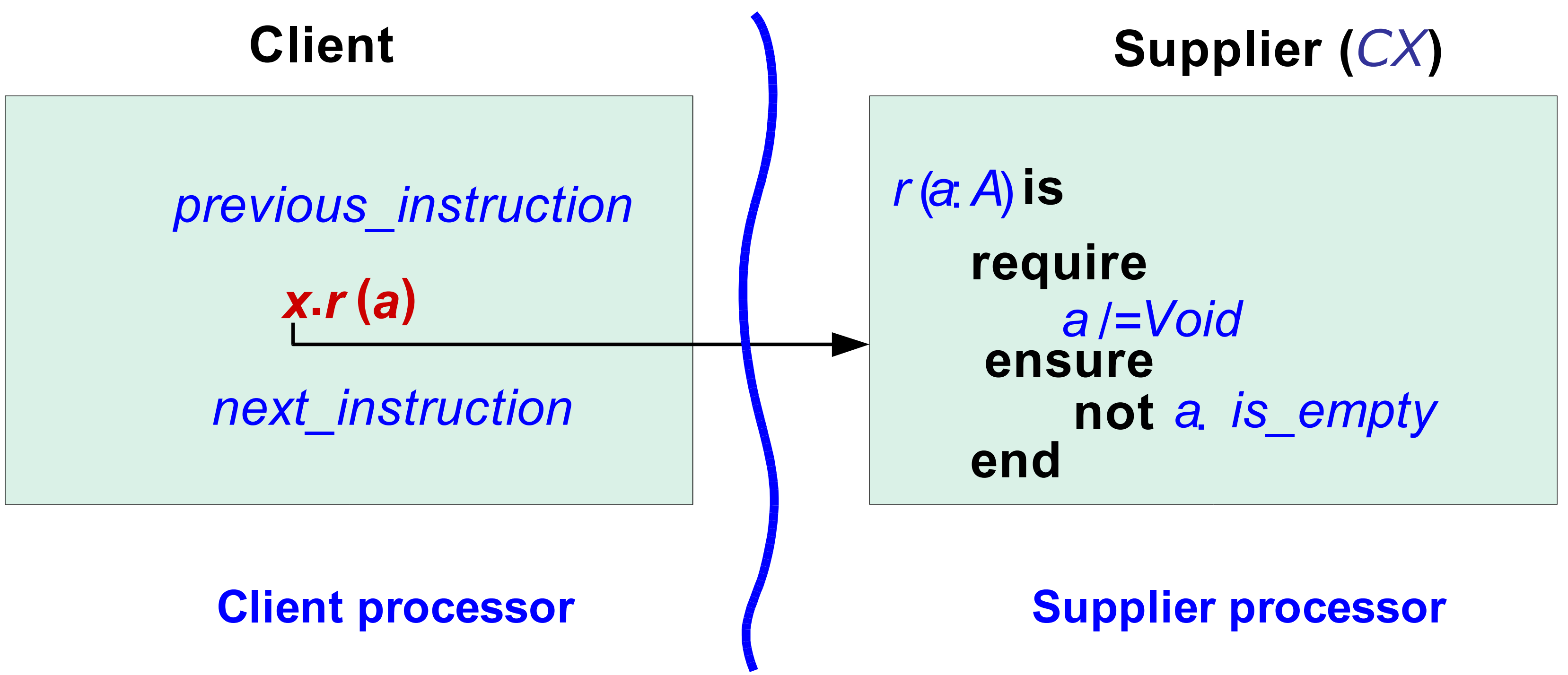
r (*a*: *A*) **is**
    require
        a /=*Void*
    ensure
        not *a*.*is_empty*
    end

Processor

Feature call: asynchronous

x.r (a)

x: **separate** *CX*



Separateness rule

Calls to non-separate objects are synchronous

Call to separate objects are asynchronous

Feature call: asynchronous

x.r (a)

x: **separate** *CX*

Client

previous_instruction

x.r (a)

next_instruction

Client processor

Supplier (*CX*)

```
r (a: A) is
    require
        a /= Void
    ensure
        not a.is_empty
    end
```

Supplier processor

Feature call: asynchronous

x.r (a)

x: **separate** *CX*

Client

previous_instruction

x.r (a) →

next_instruction

```
r (a: A) is
    require
        a /= Void
    ensure
        not
    a.is_empty
    end
```

Client processor

Supplier processor

What does "separate" mean?

- Does not specify processor
- Simply indicates that it's "elsewhere"

The fundamental difference

To wait or not to wait:

- If same processor, synchronous
- If different processor, asynchronous

Difference must be captured by syntax:

- x: CX
- x: **separate** CX

Consistency

Client:

```
class C feature
    a: SOME_TYPE
    sep: separate B
    sep.p (a)
end
```

Supplier:

```
class B feature
    p (a:                SOME_TYPE)
        is do ... end
end
```

Consistency

Client:

```
class C feature
    a: SOME_TYPE
    sep: separate B
    sep.p (a)
end
```

Supplier:

```
class B feature
    p (a: separate SOME_TYPE)
        is do ... end
end
```

Separateness consistency rule

For any reference actual argument in a separate call, the corresponding formal argument must be declared as separate

Separate call: *a.f* (...) where *a* is separate

If no access control

x: **separate** CX

…

x.r (a)

y := x.f

If no access control

```
my_stack: separate STACK [T]

…

my_stack.push (a)

 y := my_stack.top
```

Access control policy

- Require target of separate call to be formal argument of enclosing routine:

```
put (b: separate STACK [T]; value: T) is
        -- Push value on top of b.
    do
        b.push (value)
    end
```

Access control policy

- Target of a separate call must be formal argument of enclosing routine:

```
put (b: separate BUFFER [T]; value: T) is
        -- Store value into b.
    do
        b.put (value)
    end
```

- To use separate object:

```
my_buffer: separate BUFFER [INTEGER]
create my_buffer
store (my_buffer, 10)
```

Separate argument rule

The target of a separate call
must be an argument of the enclosing routine

Separate call: *a.f* (...) where *a* is separate

Wait rule

A routine call with separate arguments
will execute when all corresponding objects are available

and hold them exclusively
for the duration of the routine

Separate call: *a.f* (...) where *a* is separate

Contracts in Eiffel

```
store (buf: BUFFER [INTEGER]; value: INTEGER) is
      -- Store value into buffer.
   require
      not buf.is_full
      value > 0
   do
      buf.put (value)
   ensure
      not buf.is_empty
   end
...
```

Precondition: client obligation

Postcondition: supplier obligation

Contracts in Eiffel

```
store (buffer: BUFFER [INTEGER]; value: INTEGER) is
        -- Store value into buffer.
    require
        not buffer.is_full
        value > 0
    do
        buffer.put (value)
    ensure
        not buffer.is_empty
    end
...
store (my_buffer, 10)
```

Precondition

From preconditions to wait-conditions

```
store (buffer: separate BUFFER [INTEGER]; value: INTEGER)
  is
     -- Store value into buffer.
  require
     not buffer.is_full
     value > 0
  do
     buffer.put (value)
  ensure
     not buffer.is_empty
  end
...
store (my_buffer, 10)
```

On separate target, precondition becomes **wait condition**

- If buffer is *separate*,.

Contracts

Client:

```
if not my_buffer.is_full
then
    store (my_buffer, x)
end
```

Supplier:

```
store (b: BUFFER [T]; value: T) is
        -- Store value into b.
    require
        not b.is_full
        value > 0
    do
        b.put (value)
    ensure
        not b.is_empty
    end
...
```

Contract under concurrency?

Client:

```
if not my_buffer.is_full
   ????
then

   store (my_buffer, x)

end
```

Supplier:

```
store (b: BUFFER [T]; value: T) is
      -- Store value into b.
   require
      not b.is_full
      value > 0
   do
      b.put (value)
   ensure
      not b.is_empty
   end
...
```

What happens to preconditions?

- Precondition on separate target becomes **wait condition** (instead of correctness condition)

- This becomes the basic synchronization mechanism

Separate precondition rule

A separate precondition
causes the client to wait

Separate precondition: *a.condition* (...)
where *a* is separate

Full synchronization rule

A call with a separate argument waits until:

- Object is available
- Separate precondition holds

x.f (*a*)

where *a* is separate

Resynchronization

- No special mechanism needed for client to resynchronize with supplier after separate call.
- The client will wait only when it needs to:

x.f

x.g (a)

y.f

…

value := x.some_query

Wait here!

Resynchronization rule

Clients wait for resynchronization on queries

Interrupts?

Can we snatch shared object from its current holder?

- Execute *holder.r (b)* where *b* is **separate**
- Another object executes *challenger.s (b)*
- Normally, *challenger* would wait
- What if *challenger* is impatient?

The duel mechanism

Library features

Challenger / Holder	*normal_service*	*immediate_service*
retain	Challenger waits	Exception in challenger
yield	Challenger waits	Exception in holder; serve challenger

Extending duels

- Timing limits
- Priorities (for real-time processing)

Example: class *PROCESS*

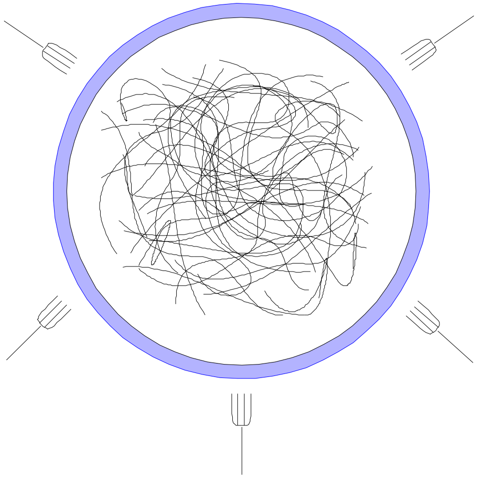
```
deferred class
    PROCESS
feature -- Status report
    over: BOOLEAN is
            -- Must execution terminate now?
        deferred   end
feature -- Basic operations
    setup is
            -- Prepare to execute process (default: nothing).
        do   end
    step is
            -- Execute basic process operations.
        deferred   end
```

PROCESS

```
    wrapup is
            -- Execute termination operations (default: nothing).
          do   end

feature -- Process behavior

    live is
            -- Perform process lifecycle.
          do
            from setup until over loop
                step
            end
            wrapup
          end
end
```

Example: Dining philosophers

```
class PHILOSOPHER inherit
    PROCESS
        rename
            setup as getup
        redefine step end

feature {BUTLER}
    step is
            do
                think ;   eat (left, right)
            end

    eat (l, r: separate FORK) is
                -- Eat, having grabbed l and r.
            do … end
end
```



Example: Bounded buffer usage

Usage of bounded buffers

```
buff: BUFFER_ACCESS [MESSAGE]
my_buffer: BOUNDED_BUFFER [MESSAGE]

create my_buffer
create buff.make (my_buffer)

buff.put (my_buffer, my_message)
…
buff.put (my_buffer, her_message)
…
my_query := buff.item (my_buffer)
```

Other examples

- Watchdog: use duels

- Elevator (see next)

- Others in *Object-Oriented Software Construction*

Duels

Problem: Impatient client (*challenger*) wants to snatch object from another client (*holder*)

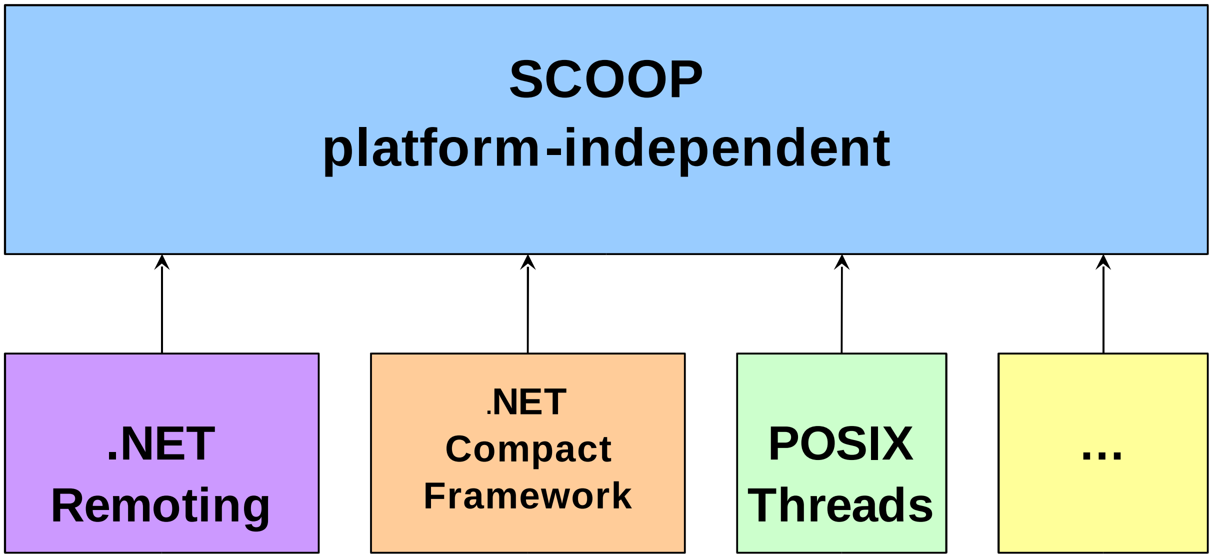
- Can't just interrupt holder, service challenger, and resume holder: would produce inconsistent object.
- But: can cause exception, which will be handled safely.

Duels

Challenger / Holder	*normal_service*	*immediate_service*
retain	Challenger waits	Exception in challenger
yield	Challenger waits	Exception in holder; serve challenger

Two-level architecture of SCOOP

- Adaptable to many environments
- .NET remoting is current platform



Mapping processors to physical resources

Concurrency Control File (CCF)

```
create
   system
      "lincoln"      (4): "c:\prog\appl1\appl1.exe"
      "roosevelt"    (2): "c:\prog\appl2\appl2.dll"
      "Current"      (5): "c:\prog\appl3\appl3.dll"
   end
external
   Database_handler: "jefferson" port 9000
   ATM_handler:      "gates"     port 8001
end
default
   port: 8001; instance: 10
end
```

SCOOPLI: Library for SCOOP

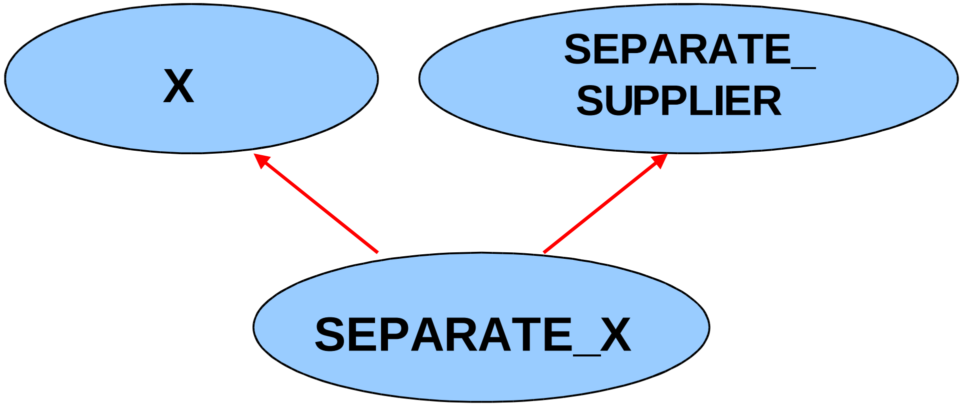
- Library-based solution
- Implemented in Eiffel for .NET
  (from Eiffel Software:
  EiffelStudio / ENViSioN! for Visual Studio.NET)
- Aim: try out solutions without bothering with compiler issues
- Can serve as a basis for compiler implementations

SCOOPLI concepts

- *separate client*
- *separate supplier*

Each separate client & separate supplier handled by different processor

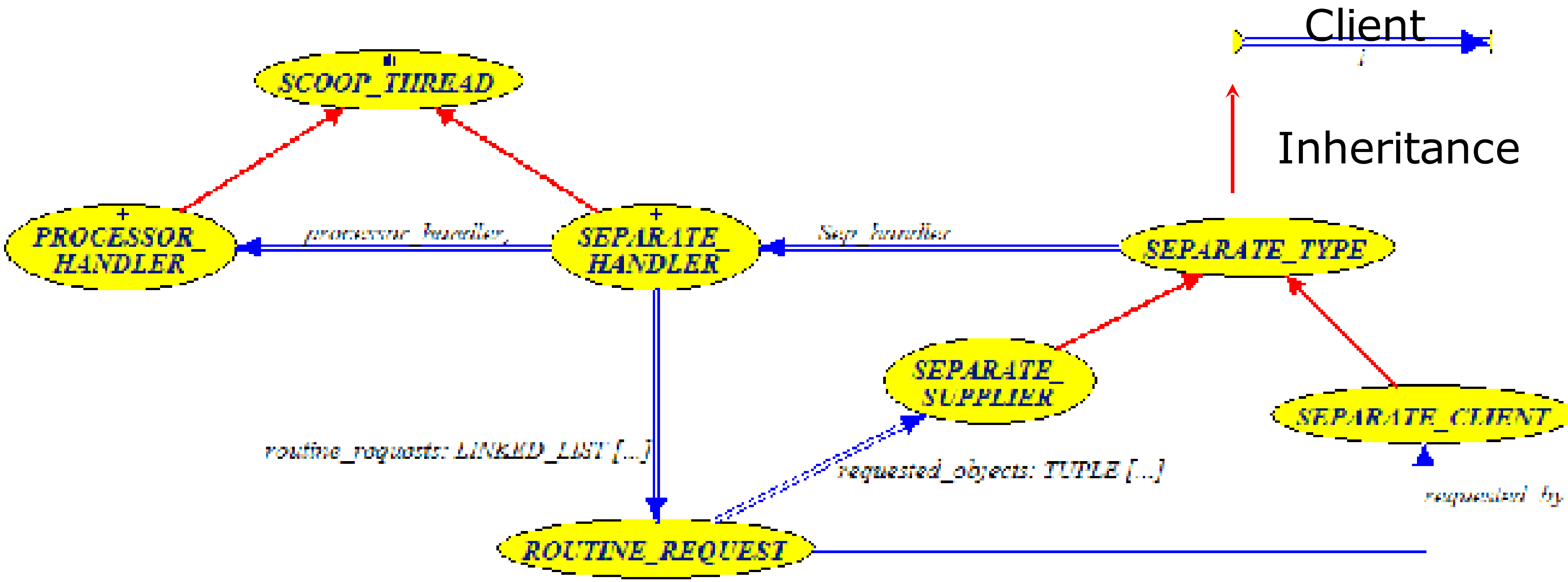
Class gets separateness through multiple inheritance:



SCOOPLI emulation of SCOOP concepts

SCOOP	SCOOPLI
x: **separate** *X* *x*: *X* -- class X is separate	*x*: *SEPARATE_X* -- *SEPARATE_X* inherits from *X* and -- *SEPARATE_SUPPLIER*
r (*x*, *y*) -- x and y are separate *r* (*x*: **separate** *X*; *y*: **separate** *Y*) **is** **require** **not** *x.is_empty* *y.count* > 5 *i* > 0 -- i non-separate x /= Void **do**	separate_execute ([*x*, *y*], **agent** *r* (*x*, *y*), **agent** *r_precondition*) *r_precondition*: BOOLEAN **is** **do** **Result** := **not** *x.is_empty* **and** *y.count* > 5 **end** -- client class inherits from -- class SEPARATE_CLIENT

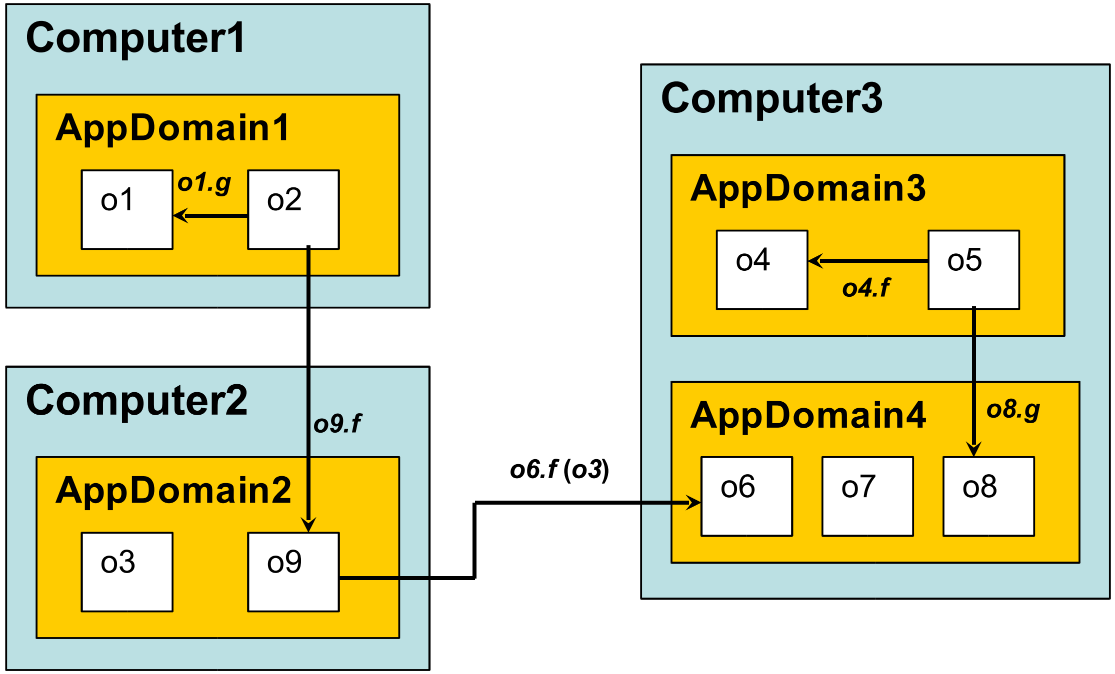
SCOOPLI Architecture



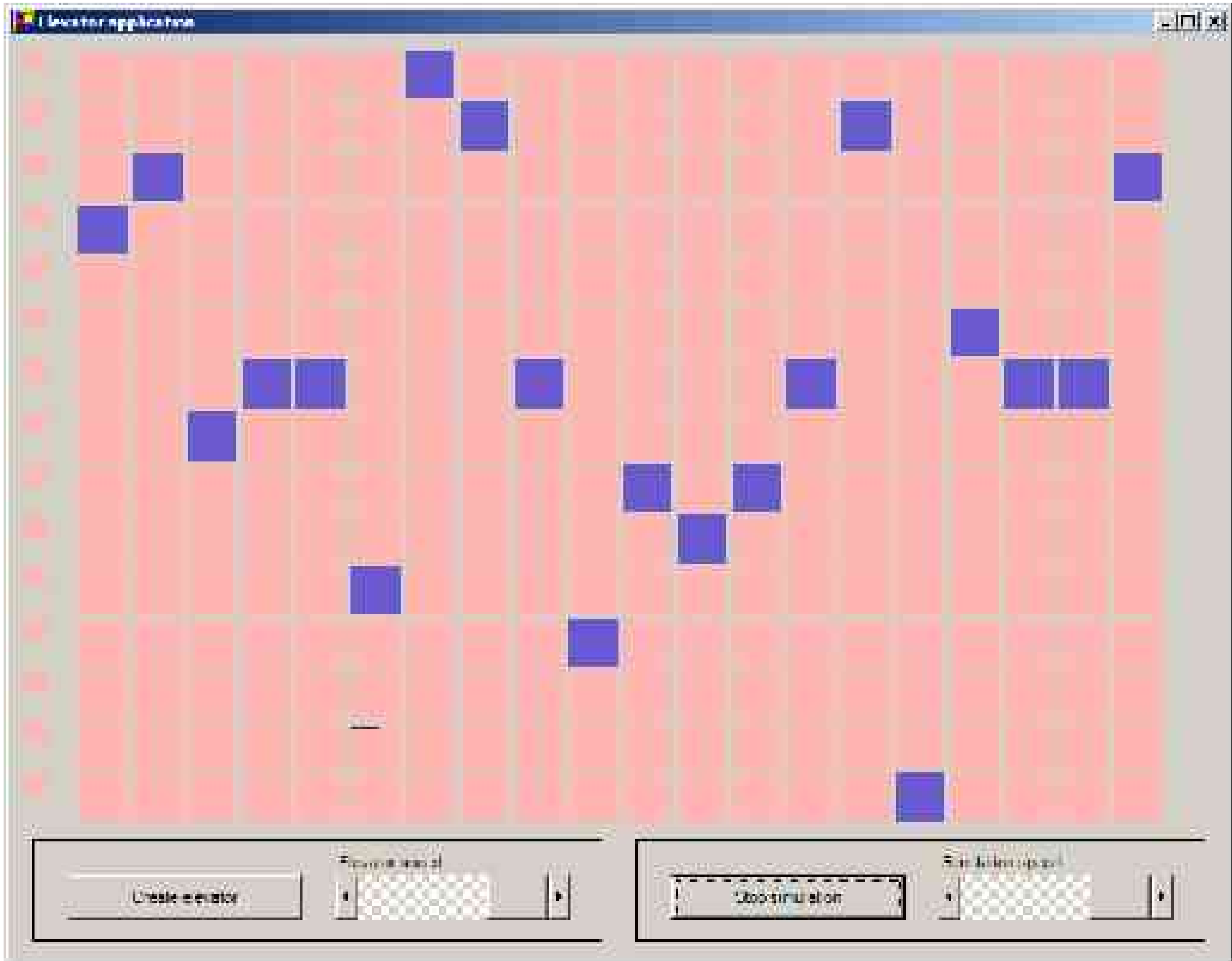
- *SEPARATE_HANDLER*: locking; checking wait conditions; scheduling of requests
- *PROCESSOR_HANDLERs*: execute separate calls; implement processors

Distributed execution

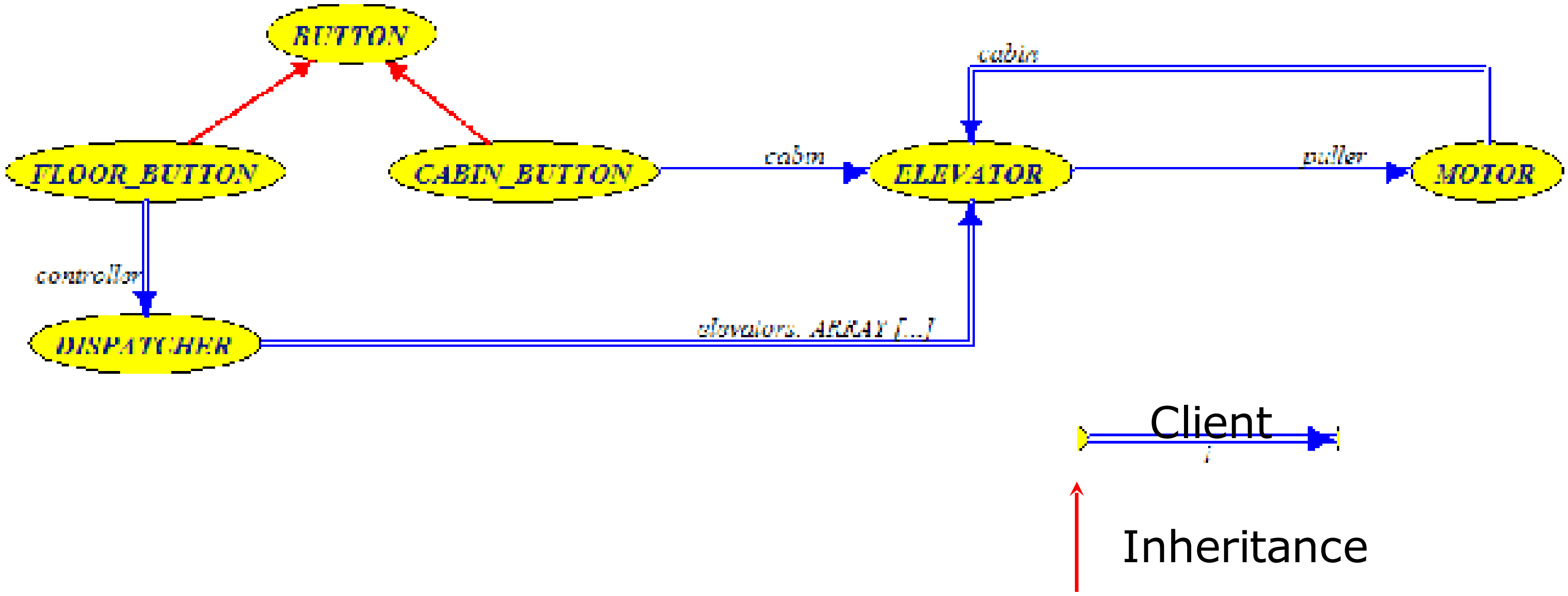
- Processors (AppDomains) located on different machines
- .NET takes care of the "dirty work"
  - Marshalling
  - Minimal cost of inter-AppDomain calls



SCOOP multithreaded elevators



Elevator example architecture



For maximal concurrency, all objects are separate

Class BUTTON

```
separate class
    BUTTON
feature
    target: INTEGER
end
```

Class *CABIN_BUTTON*

```
separate class CABIN_BUTTON inherit
    BUTTON
feature
    cabin: ELEVATOR

    request is
            -- Send to associated elevator a request to stop on level target.
        do
            actual_request (cabin)
        end

    actual_request (e: ELEVATOR) is
            -- Get hold of e and send a request to stop on level target.
        do
            e.accept (target)
        end
end
```

Class *ELEVATOR*

```
separate class ELEVATOR feature {BUTTON, DISPATCHER}

   accept (floor: INTEGER) is
         -- Record and process a request to go to floor.
      do
         record (floor)
         if not moving then process_request end
      end

feature {MOTOR}

   record_stop (floor: INTEGER) is
         -- Record information that elevator has stopped on
   floor.
      do
         moving := False ; position := floor ; process_request
      end
```

Class *ELEVATOR*

```
feature {NONE} -- Implementation
    process_request is
            -- Handle next pending request, if any.
        local floor: INTEGER do
            if not pending.is_empty then
                floor := pending.item ; actual_process (puller, floor)
                pending.remove
            end
        end

    actual_process (m: MOTOR; floor: INTEGER)  is
            -- Handle next pending request, if any.
        do
            moving := true ; m.move (floor)
        end

feature {NONE} -- Implementation
    puller: MOTOR ; pending: QUEUE [INTEGER]
end
```

Class *MOTOR*

```
separate class MOTOR feature {ELEVATOR}
    move (floor: INTEGER) is
            -- Go to floor; once there, report.
        do
            gui_main_window.move_elevator (cabin_number, floor)
            signal_stopped (cabin)
        end
    signal_stopped (e: ELEVATOR) is
            -- Report that elevator e stopped on level position.
        do  e.record_stop (position) end
feature {NONE}
    cabin: ELEVATOR   ; position: INTEGER    -- Current floor level.
    gui_main_window: GUI_MAIN_WINDOW
end
```

Why SCOOP?

- SCOOP model
  - Simple yet powerful
  - Easier and safer than common concurrent techniques, e.g. Java Threads
  - Full concurrency support
  - Full use O-O and Design by Contract
  - Supports various platforms and concurrency architectures
  - One new keyword: **separate**

- SCOOPLI library
  - SCOOP-based syntax
  - Implemented on .NET
  - Distributed execution with .NET Remoting

Future work & open problems

- Other “handles”
- Direct support for distribution
- Prevent deadlock, extend access control policy
- Extend for real-time
  - Duel mechanism with priorities
  - Timing assertions?
- Integrate with Eiffel Software compiler

Application to Web services

- Every Web service should be described by a contract
- SCOOP seems to provide the right conceptual framework
- Implementation is in progress

Why SCOOP?

- Extend object technology with general and powerful concurrency support

- Provide the industry with simple techniques for parallel, distributed, internet, real-time programming

- Make programmers sleep better!