ETH Zürich
Chair of Software Engineering

# Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

## Lecture 6: SCOOP principles

# SCOOP mechanism

*Simple Concurrent Object-Oriented Programming*

Evolved through last decade; *CACM* (1993) and chap. 32 of *Object-Oriented Software Construction*, 2nd edition, 1997

Protoype-implementation at ETH

Ongoing integration into EiffelStudio by EiffelSoftware

# SCOOP preview: a sequential program

```
transfer (source, target:                ACCOUNT;
            amount: INTEGER)
        -- If possible, transfer amount from source to target.
    do
        if source.balance >= amount then
            source.withdraw  (amount)
            target.deposit   (amount)
        end
    end
```

Typical calls:

```
transfer (acc1, acc2, 100)
transfer (acc1, acc3, 100)
```

# In a concurrent setting, using SCOOP

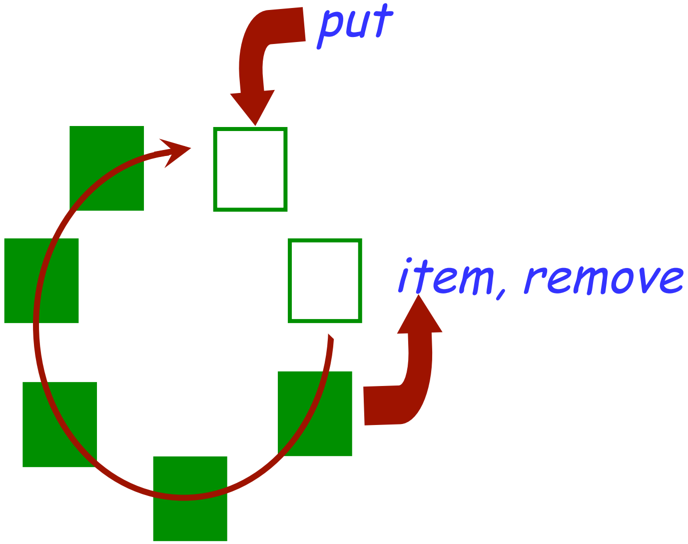
```
transfer (source, target: separate ACCOUNT;
          amount: INTEGER)
        -- If possible, transfer amount from source to target.
    do
        if source.balance >= amount then
            source.withdraw  (amount)
            target.deposit   (amount)
        end
    end
```

Typical calls:

```
transfer (acc1, acc2, 100)
transfer (acc1, acc3, 100)
```

# A better SCOOP version

```
transfer (source, target: separate ACCOUNT;
          amount: INTEGER)
        -- Transfer amount from source to target.
    require
         source.balance >= amount
    do
        source.withdraw  (amount)
        target.deposit   (amount)
    ensure
        source.balance = old source.balance – amount
        target.balance = old target.balance + amount
    end
```




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

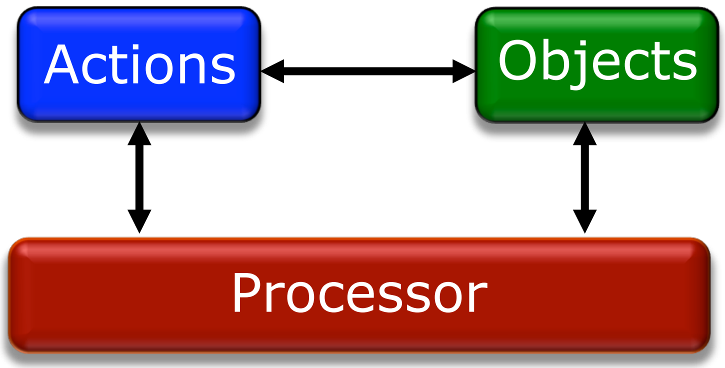
```
my_queue : BUFFER QUEUE [T ]
...

if not my_queue.is_full then



    put (my_queue, t )
end
```

# Processors in SCOOP

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

Can be implemented as:

- Computer CPU
- Process
- Thread




Will be mapped to computational resources.

# Handler rule

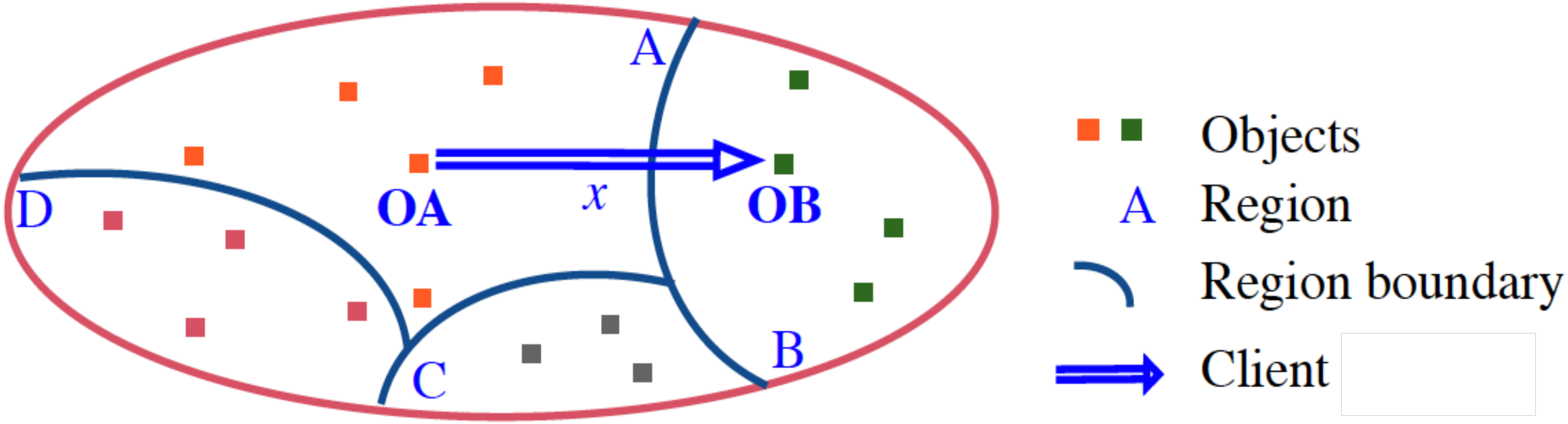
- The computational model of SCOOP relies on the following fundamental rule:

All calls targeted to a given object are performed by a single processor, called the object's *handler*.

- A call is "targeted" to an object in the sense of object-oriented programming: the call *x.r* applies the routine *r* to the *target* object identified by *x*.

# Regions

- The set of objects handled by a given processor is called a *region*.
- The Handler rule implies a one-to-one correspondence between processors and regions.

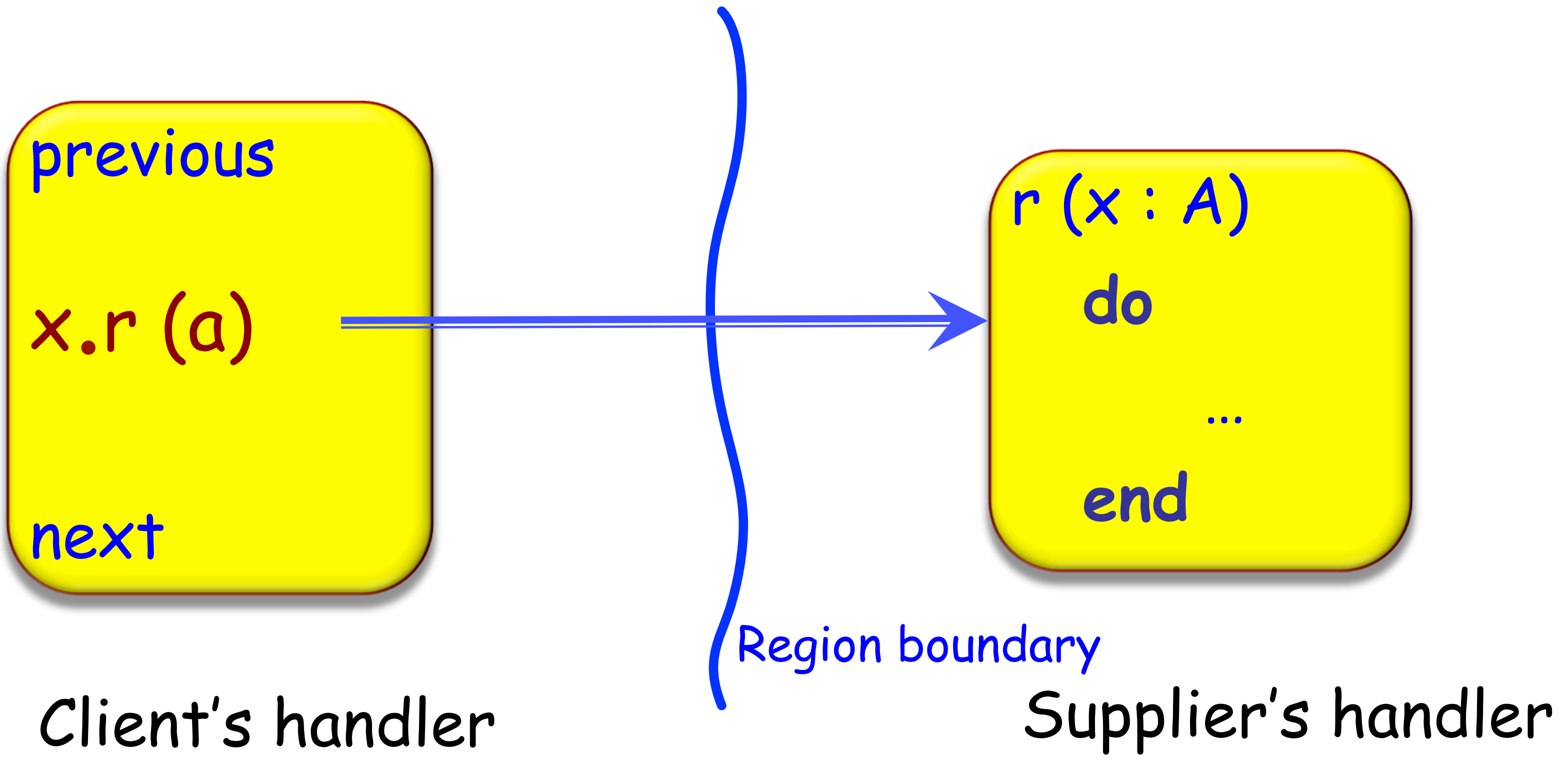


## Separate declarations

- SCOOP introduces the keyword **separate**, which is a type modifier
- If *x* is declared **separate** T for some type T, then the associated object will normally be handled by a different processor.
- For example, if a processor *p* executes a call *x.r*, and *x* is handled by processor *q*, then *q* (rather than *p* itself) will execute *r*.
- Terminology: a call *x.r* is a *separate call* if its target *x* is separate.
- The usual semantics remains: If *x* is declared as just *T*, not separate, the current processor *p* will execute *r*.

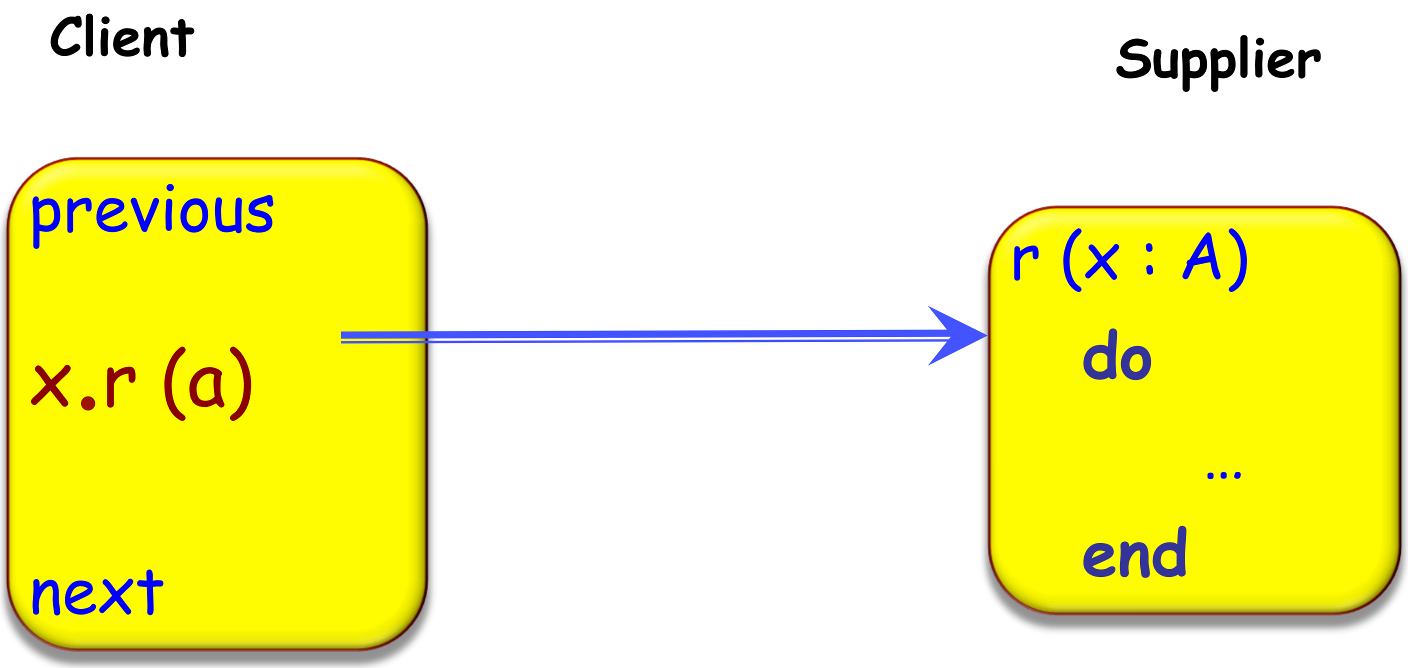
# Separate call (asynchronous)

- Separate calls are executed *asynchronously*:
  - A client executing separate call x.r(a) logs the call with the handler of x (who will execute it)
  - The client can proceed executing the next instructions without waiting



# Ordinary call (synchronous)

- With non-separate calls, the semantics is the same as in sequential computation
- The client waits for the call to finish (synchronous)



# Routine call and routine application

- The introduction of asynchrony highlights a difference between two notions:
  - A routine *call*, such as *x.r* executed by a certain processor *p*.
  - A routine *application*, which — following a call — is the execution of the routine *r* by a processor *q*.
- While the distinction exists in sequential programming, it is especially important in SCOOP, as processors *p* and *q* might be different from each other.

# Summary: the fundamental difference

To wait or not to wait:

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

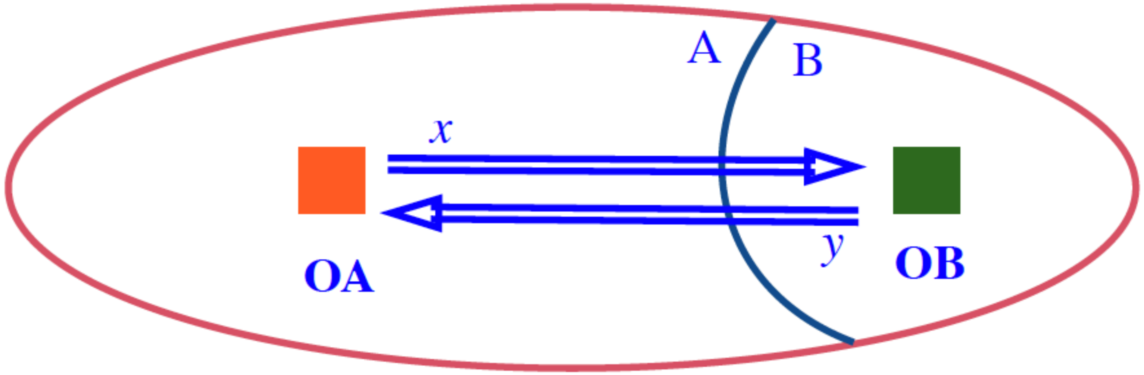
Difference is captured by type system:

- x: T
- x: **separate** T -- *Potentially* different processor

Fundamental semantic rule: *x*.*r* (*a*) waits for non-separate *x*, doesn't wait for separate *x*.

# Why *potentially* separate?

- A **separate** declaration does not specify the processor; it only specifies that the corresponding object *might* be handled by a processor that is not the same as the current object's handler.
    - In class A: *x*: **separate** *B*
    - In class B: y: **separate** *A*
    - In some execution the value of x.y might be a reference to an object handled by the current object, or even the current object itself.



# Lazy wait (1)

- What if a client needs to resynchronize with a separate object on which you have launched a separate call?

  *x.f*

  *x.g (a)*

  *y.f*

  …

  *value := x.some_query*

- In SCOOP, we resynchronize only on *queries* – the client only waits if it needs to (*lazy wait*)
- Recap:
  - A *command* does not return a result (procedure).
  - A *query* returns a result (function or attribute).

## Lazy wait (2)

- Lazy wait changes the rule for separate calls as follows:
    - A processor executing a separate call to a query will not proceed until the result of the query has been computed.
    - For a separate call to a command, the processor can proceed without waiting as soon as it has logged the call.
- Lazy wait is also called wait by necessity (D. Caromel).

# Mutual exclusion in SCOOP

- SCOOP has a simple way to express mutual exclusive access to objects by way of *argument passing*
- The SCOOP runtime system makes sure that the application of a call *x.r (a1, a2, ...)* will *wait* until it has been able to lock *all* the separate objects associated with the arguments *a1, a2, ...* .
- Within the routine body, the access to the separate objects associated with the arguments *a1, a2, ...* is thus mutually exclusive.
- Note that in difference to other formalisms, SCOOP thus provides a simple way to lock a *group of objects* at the same time.

# Example: Mutual exclusion

- For example, in the execution of the following routine we can rely on the runtime system to lock the separate argument b:

```
put (b: separate QUEUE[T]; value: T)
        -- Add value, FIFO-style, to b.
    do
        b.put (value)
    end
```

- Hence the modification of the buffer *b.put (value)* will be executed safely (in mutual exclusion with other accesses)

# Example: dining philosophers in SCOOP

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

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

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

# The separate argument rule

- Argument passing is *enforced* in SCOOP, to protect modifications on separate objects
- The following rule expresses this:

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

- For example the following code would give an compile time error since *b* is not an argument of *put*:

```
b: separate QUEUE[T]
put (value : T)
        do
            b.put (value)
        end
```

# Condition synchronization in SCOOP

- Condition synchronization is provided in SCOOP by reinterpreting routine *preconditions* as *wait conditions*.
- This means that the execution of the body of a routine is delayed until its separate preconditions are satisfied
- A *separate precondition* is a precondition that involves a call to a separate target.

```
put (buf : separate QUEUE[INTEGER] ; v : INTEGER)
        -- Store v into buffer.
    require
        not buf.is_full
        v > 0
    do
        buf.put (v)
    ensure
        not buf.is_empty
    end
```

Correctness condition (no wait semantics)

Precondition becomes **wait condition**

# Wait rule

- The behavior of the SCOOP runtime system with respect to waiting for a routine application is summarized in the following rule:

> A call with separate arguments waits until the corresponding objects' handlers are all available, and the separate conditions all satisfied. It reserves the handlers for the duration of the routine's execution.

# SCOOP runtime system: request queues

- When a processor makes a separate feature call, it sends a feature request.
- Each processor has a request queue to keep track of these feature requests.

```
test (a_buffer: separate BUFFER [INTEGER])
        -- Test the buffer 'a_buffer'.
    require
        a_buffer_is_empty: a_buffer.count = 0
    local
        I: INTEGER
    do
        a_buffer.put (2)
        a_buffer.put (6)
        I := a_buffer.item
        I := a_buffer.item
    end
```

buffer processor request queue:

# SCOOP runtime system: scheduler

- Before a processor can process a *feature request* it must:
  - Obtain the necessary locks
  - Satisfy the precondition
- The processor sends a *locking request* to a scheduler.
- The scheduler keeps track of the locking request. It approves locking requests according to a scheduling algorithm.
- Several scheduling algorithms are possible:
  - Centralized vs. decentralized
  - Different levels of fairness

# SCOOP runtime system: separate callbacks

```
class CONSUMER ...
        id: INTEGER

        check_id (a_buffer: separate BUFFER [INTEGER])
                -- Check whether 'a_buffer' has the consumer's identifier.
            local
                l: BOOLEAN
            do
                l := a_buffer.has_id (Current)
            end
end

class BUFFER [G] ...
        has_id (a_consumer: separate CONSUMER): BOOLEAN
                -- Is the identifier of 'a_consumer' in the buffer?
            do
                Result := area.has (a_consumer.id)
            end
end
```

The consumer processor waits for the query to return.

deadlock

Separate callback: the buffer processor waits for the query to return.

# SCOOP runtime system: separate callbacks

- Solution:
  - The buffer processor interrupts the consumer processor from waiting.
  - The buffer processor asks the consumer processor to execute the feature request right away.
- How to detect a separate callback?
  - The consumer processor has a lock on the buffer processor.
  - This means that the consumer processor is (potentially) waiting for the buffer processor.
  - The buffer processor can detect this at the moment of the separate callback.

# What can SCOOP do for us?

Beat enemy number one in concurrent world: atomicity violations

- Data races
- Illegal interleaving of calls

Data races cannot occur in SCOOP

- Why? See computational model ...

Separate call rule does not protect us from bad interleaving of calls!

- How can this happen?

# Why SCOOP?

- Simple (one new keyword) yet powerful
- Easier and safer than common concurrent techniques, e.g. Java Threads
- Full concurrency support
- Full use of O-O and Design by Contract
- Retains ordinary thought patterns, modeling power of O-O
- Supports  wide range of platforms and concurrency architectures
- Programmers need to sleep better!