Object-Oriented Software Construction

Bertrand Meyer

Lecture 19: SCOOP
Simple Concurrent Object-Oriented Programming

Piotr Nienaltowski

What we're going to see today

- Some history
- Back to the roots: the essence of OO
- SCOOP: computational model
- Synchronisation
- Advanced mechanisms
- How does it fit into the OO framework?

The need for concurrency: where

- Operating systems
  - We already forgot the "batch processing" systems from the 60s
  - Users want multi-tasking
- Distributed applications
- Modern GUIs
- Many applications only make sense in the presence of concurrency
- We want to squeeze maximum power from our computers
  - Use additional CPUs

The need for concurrency: how

- Two basic kinds of concurrency
  - Cooperative
  - Competitive
- Cooperative
  - several tasks want to achieve a common goal
  - synchronisation is necessary to achieve the goal
  - example: producer-consumer
- Competitive
  - several task access resources to do their own job
  - no common goal; tasks often don't know about each other
  - synchronisation is necessary to avoid conflicts
  - example: flight booking system

The need for concurrency: what

- Two kinds of properties of interest
  - Safety (nothing bad happens)
  - Liveness (something good eventually happens)
- Data races
- Deadlocks
- Livelocks

A bit of history

- Mutex
- Semaphore (60s - Dijkstra)
- Conditional Critical Region (1968, Hoare)
- Monitor (1972, Brinch-Hansen, Hoare)
- Rendez-vous (80s, Ada), active objects

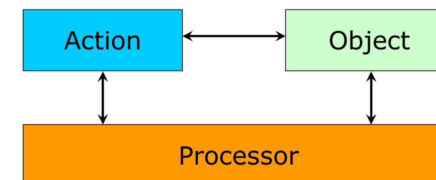
SCOOP

- Simple Concurrent Object-Oriented Programming
- Described in CACM, 1993
- OOSC, 2nd edition, 1997
- Prototype implementation at Eiffel Software, 1995
- Prototypes by others
- Now being done for good at ETH (support from SNF, Hasler Foundation, Microsoft ROTOR)

Basic idea of OO computation

To perform a computation is
- To apply certain actions
- To certain objects
- Using certain processors

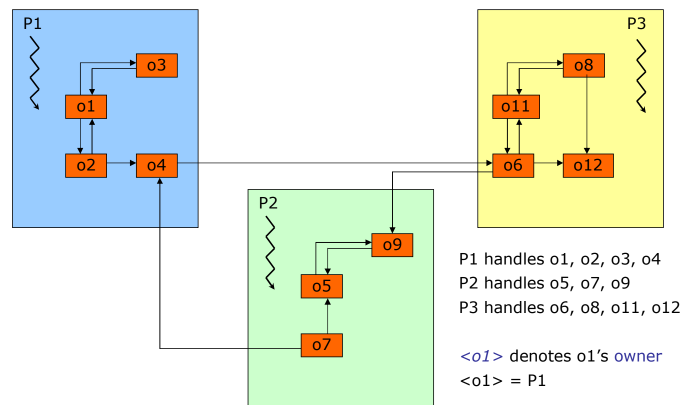
x.f (a)



Computational model of SCOOP

- Processor: a thread of control supporting sequential execution of instructions on one or several objects.
- All actions on a given object are executed by its handling processor. No shared memory!!!
- We say that an object is owned by its handling processor
  - this ownership relation is fixed, i.e. we do not consider migration of objects between processors.
- Each processor, together with all object it owns, can be seen as a sequential subsystem.
- A (concurrent) software system is composed of such subsystems.

Software system



Processors

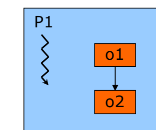
- Processor is an abstract concept
- Do not confuse it with a CPU!
- A processor can be implemented as:
  - Process
  - Thread
  - Web service
  - .NET AppDomain
  - ???

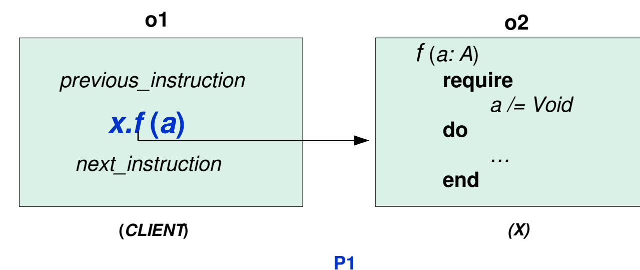
Feature call - synchronous

x: *X*

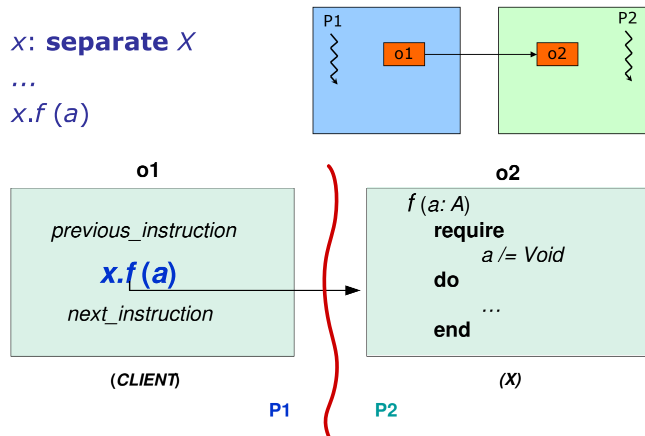
...

x.*f* (*a*)



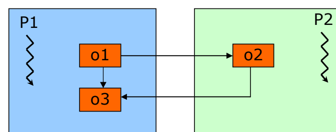


Feature call - asynchronous



Separate objects

- Calls to non-separate objects are synchronous
- Calls to separate objects are asynchronous



- QUIZ: Which objects are separate?

Separate entities

- Separate entities are declared with **separate** keyword

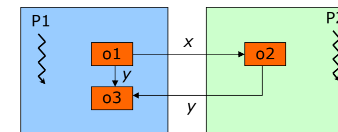
  x: **separate** *X*

- Does a separate entity always denote a separate object?

  x, *y*: **separate** *X*

  ...

  y := *x.y* -- Is *y* a separate entity?
  -- Does it denote a separate object?



- Separate entities denote potentially separate objects

Synchronisation

- Processors are sequential
- Concurrency is achieved by interplay of several processors
- Processors need to synchronise
- Two forms of synchronisation in SCOOP
  - mutual exclusion
  - condition synchronisation

If no mutual exclusion

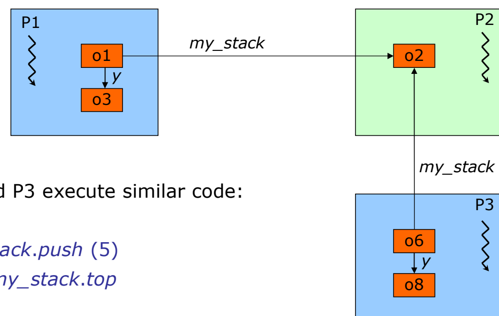
- Programmer writes:

```
my_stack: separate STACK [INTEGER]

...

my_stack.push (5)
y := my_stack.top    -- Are we sure that y = 5 ?
```

What could have happened here?

We need a critical section to avoid data races.

Problematic scenario



P1 and P3 execute similar code:

```
-- P1
my_stack.push (5)
y := my_stack.top

-- P3
my_stack.push (100)
y := my_stack.top
```


Mutual exclusion in SCOOP

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

```
push_and_retrieve (s: separate STACK [INTEGER];
                value: INTEGER) is
        -- Push `value' on top of `s' then retrieve top of `s'
        -- and assign it to `y'.
    do
        s.push (value)
        y := s.top
    end
```

No other processor can access *s* in the meantime!

```
my_stack: separate STACK [INTEGER]
...
push_and_retrieve (my_stack, 5) -- Now we are we sure that y=5
```

- Body (do ... end) of enclosing routine is a critical section with respect to its separate formal arguments.

Separate argument rule

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

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

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

Condition synchronisation

- Very often client only wants to execute certain feature if some condition (guard) is true:

```
store (buffer: separate BOUNDED_BUFFER [INTEGER];
        value: INTEGER) is
            -- Store `value' into `buffer'.
    require
            buffer_not_full: not buffer.is_full
    do
            buffer.put (value)
    end

my_buffer: separate BOUNDED_BUFFER [INTEGER]
...
store (my_buffer, 5)
```

Hey, it's a precondition, not a guard!

How should it work?

Contracts strike back ☺


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

...
store (my_buffer, 10)
```

Precondition

Postcondition

From preconditions to wait-conditions


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

...
store (my_buffer, 10)
```

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

Why new semantics?

- Preconditions are obligations that client has to satisfy before the call

  $\{Pre_r\}$ **call** r $\{Post_r\}$

- Easy peasy:

  if *precondition*$_{store}$ **then**
  store (*my_buffer*, 5)
  end

  ~~I know that precondition holds before the call!~~

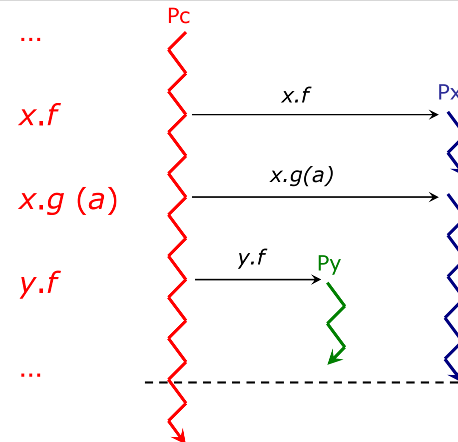
Wait rule revisited

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

and wait-conditions are satisfied

and hold the objects exclusively for the duration of the routine

Re-synchronising clients and suppliers



Wait by necessity

- No special mechanism for client to re-synchronise with supplier after separate call.
- Client will only wait when it needs to:

  x.f
  x.g (*a*)
  y.f
  …
  value := *x.some_query* — Wait here!

- This is called **wait-by-necessity**

Do we *really* need to wait?

- Can we do better than that?

  x.f
  x.g (a)
  y.f
  …
  value := x.some_query
  x.f
  y.f
  z := value
  value := value + 1

  We only need to wait here!

- Does not change the basic SCOOP model
- Consider it to be an optimisation

Summary: computational model

- Software system is composed of several **processors**
- Processors are **sequential**; concurrency is achieved through their interplay
- Separate entity denotes a *potentially* **separate object**
- Calls to non-separate objects are **synchronous**
- Calls to separate objects are **asynchronous**

Summary: synchronisation

- Mutual exclusion
  - Locking through **argument passing**
  - Routine **body** is **critical section**
- Condition synchronisation
  - **preconditions**
- Re-synchronisation of client and supplier:
  - **wait-by-necessity**

Summary: separate argument rule

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

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

Summary: wait rule

33

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

and wait-conditions are satisfied

and hold the objects exclusively for the duration of the routine

Example: bounded buffer

34

Now that we know (almost) everything, let's see a short example

Is that enough to prevent data races?

35

- *Data race occurs when two or more clients concurrently apply some feature on the same supplier.*
- Data races could be caused by so-called traitors, i.e. non-separate entities that denote separate objects.
  - Kill 'em all!

Traitors

36

```
-- in class C (client)
x: separate X
a: A
...
r (an_x: separate X) is
   do
      a := an_x.a
   end
...
r (x)
a.f
```

```
-- supplier
class X
feature
   a: A
end
```

Is this call valid?

TRAITOR! TRAITOR!

And this one?

Consistency rules – first attempt

- Four consistency rules
- Should prevent data races
  - eliminate traitors
- Written in English
- Easy to understand by programmers

SCOOP rules – first attempt

Separateness consistency rule (1)
If the source of an attachment (assignment instruction or argument passing) is separate, its target entity must be separate too.

```
r (buffer: separate BUFFER [X]; x: X ) is
    local
        b1: separate BUFFER [X]
        b2: BUFFER [X]
        x2: separate X
    do
        b1 := buffer -- valid
        b2 := b1     -- invalid
        r (b1, x2)   -- invalid
    end
```


SCOOP rules – first attempt

Separateness consistency rule (2)
If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.

```
store (buffer: separate BUFFER [X]; x: X ) is
    do
        buffer.put (x)
    end

-- in class BUFFER [G]
put (element: separate G) is
    . . .
```


SCOOP rules – first attempt

Separateness consistency rule (3)
If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.

```
consume_element (buffer: separate BUFFER [X]) is
    local
        element: separate X
    do
        element := buffer.item
        . . .
    end
-- in class BUFFER [G]
item: G is
    . . .
```

SCOOP rules – first attempt

Separateness consistency rule (4)

If an actual argument of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

```
store (buffer: separate BUFFER [X]; x: X ) is
    do
        buffer.put (x)      -- X must be "fully expanded"
    end

-- in class BUFFER [G]
put (element: G) is       -- G is not declared as separate anymore
    . . .
```


Consistency rules – second attempt

- What can a type system do better than a set of informal rules?
- Prevent data races
  - static (compile-time) checks
- Integrate expanded types and agents with SCOOP
- Ownership-like types
  - Eiffel types augmented with *owner tags*
  - inspired by Peter Mueller's work on applet isolation in JavaCard
- **Subtyping rule** replaces all previous consistency rules

Type system

Let *TypeId* denote the set of declared type identifiers of a given Eiffel program. We define the set of tagged types for a given class as

$$TaggedType = OwnerId \times TypeId$$

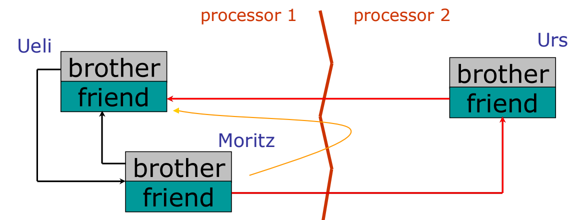
where *OwnerId* is a set of owner tags declared in the given class. Each class implicitely declares two owner tags: • (*current processor)* and ⊥ (*undefined)*.

The subtype relation $\prec$ on tagged types is the smallest reflexive, transitive relation satisfying the following axioms, where α is a tag, *S*,*T* ∊ *TypeId*, and $\prec_{Eiffel}$ denotes the subtype relation on *TypeId*:

$$(\alpha,T) \prec (\alpha,S) \Leftrightarrow T \prec_{Eiffel} S$$

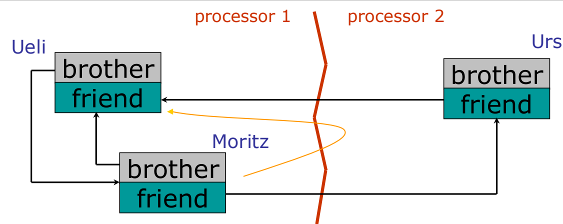
$$(\alpha,T) \prec (\perp,T)$$

False traitors




```
meet_someone_elses_friend (person: separate PERSON) is
    local
        a_friend: PERSON
    do
        a_friend := person.friend -- Invalid assignment.
        visit (a_friend)
    end
```

Handling false traitors




```
meet_someone_elses_friend (person: separate PERSON) is
    local
        a_friend: PERSON
    do
        a_friend ?= person.friend -- Valid assignment attempt.
        if a_friend /= void then visit (a_friend) end
    end
```


Duels

Library features

holder \ challenger	normal_service	immediate_service
retain	challenger waits	exception in challenger
yield	challenger waits	exception in holder; serve challenger

Interrupts?

Can we snatch a 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?

How does it fit within the OO?

- Basic mechanism: feature call
- Design by Contract
  - Generalised semantics for preconditions, postconditions, and invariants
- Inheritance
  - No particular restrictions, usual rules apply
  - Most inheritance anomalies eliminated!
- Genericity: full support
- Agents: almost full support
  - Unclear semantics of agents with open target

Genericity

49

my_array: *ARRAY* [*X*]

my_array: **separate** *ARRAY* [*X*]

my_array: *ARRAY* [**separate** *X*]

my_array: **separate** *ARRAY* [**separate** *X*]

Chair of Software Engineering — OOSC – Summer Semester 2005 — ETH

Problem: agents

50

```
a: PROCEDURE [ANY, TUPLE]

store (buffer: separate BUFFER [X]; x: X ) is
    do
        buffer.put (x)
        a := agent buffer.put (?)      -- Valid!
        fishy_call (x)
    end

fishy_call (x: X) is
    do
        a.call ([x])   -- Valid! But a is a traitor!
    end
```

Chair of Software Engineering — OOSC – Summer Semester 2005 — ETH

Solution: separate agents

51

```
a: separate PROCEDURE [ANY, TUPLE]

store (buffer: separate BUFFER [X]; x: X ) is
    do
        buffer.put (x)
        a := agent buffer.put (?)      -- separate agent
        fishy_call (x)
    end

fishy_call (x: X) is
    do
        a.call ([x])   -- Invalid: a is not a formal argument.
    end
```

Chair of Software Engineering — OOSC – Summer Semester 2005 — ETH

Separate agents

52

```
a: separate PROCEDURE [ANY, TUPLE]

store (buffer: separate BUFFER [X]; x: X ) is
    do
        buffer.put (x)
        a := agent buffer.put (?)
        not_so_fishy_call (a, x)
    end

not_so_fishy_call (an_agent: separate PROCEDURE [ANY, TUPLE]; x: X) is
    do
        an_agent.call ([x])   -- Valid: a is a formal argument.
    end
```

different semantics for argument passing: lock *an_agent.target*

Chair of Software Engineering — OOSC – Summer Semester 2005 — ETH

Additional stuff

Damn! I was very fast and I need to fill the remaining time with something interesting.

Do you want to see another example or go home?

That's all, folks!

Thanks!

And don't forget to SCOOP it up!