SIMPLE CONCURRENT OBJECT-ORIENTED PROGRAMMING

Bertrand Meyer

http://eiffel.com

SUPPORTING MATERIAL

See chapter 32 of

***Object-Oriented Software Construction, second edition*, Prentice Hall, 1997**

where this discussion is complemented by its extensions to persistence and object-oriented databases.

See: http://eiffel.com

(File doc/oosc.html)

PLAN

1. The question.

2. The constraints.

3. A solution.

4. Example sketches.

THE GOAL

Provide a simple, general, easy to use concurrency and distribution mechanism for programming concurrent applications:

- **Internet and Web programming.**
- **Client-server applications.**
- **Distributed processing.**
- **Multi-threading.**
- **Multiple processes (Unix, Windows 95, Windows NT).**

THE QUESTION

What is the simplest extension of object technology that will support all forms of concurrent computation — in an elegant, general and efficient way?

THE BASIC MECHANISM OF OBJECT-ORIENTED COMPUTATION

Feature call (message passing):

***x* ✍ *f* (*a*)**

TYPES OF CONCURRENCY

Internet programming

Threads (e.g. Posix, Solaris, Java)

Unix / Windows processes

Local network

Coroutines

CONCURRENT O-O PROGRAMMING SHOULD BE EASY!

(BUT: IT'S NOT.)

Analogies between objects/classes and processes/process-types:

1• General decentralized structure, independent modules.

2• Encapsulated behavior (a single cycle for a process; any number of routines for a class).

3• Local variables (attributes of a class, variables of a process or process type).

4• Persistent data, keeping its value between successive activations.

BUT THE ANALOGY BREAKS DOWN QUICKLY...

... and leaves room to apparent incompatibilities:

- **Classes are repositories of services; it is fundamental that they should be able to support more than one.**
- **How will processes serve each other's requests?**
- **The "inheritance anomaly"**

CAPTURING COMMON BEHAVIORS

```
deferred class PROCESS feature
  live is
      -- General structure with variants.
    do
      from setup until over loop
        step
      end
      finalize
    end
```



```
  feature {NONE}
    setup is deferred end

    over: BOOLEAN is deferred end

    step is deferred end

    finalize is deferred end
  end
```

Why limit ourselves to just one behavior when we can have as many as we want?

A PRINTER MECHANISM

```
class PRINTER inherit
  PROCESS
      rename over as off_line, finalize as stop end
feature

  stop is
        -- Go off-line.
      do off_line := true end
feature

  step is
        -- Execute individual actions of an iteration step.
      do
        start_job; process_job; finish_job
      end
```

A PRINTER MECHANISM (Continued)

```
feature {NONE}
  setup is
      do ... end
  start_job is
      do ... end
  process_job is
      do ... end
  finish_job is
      do ... end
end
```


OTHER POSSIBLE FEATURES:

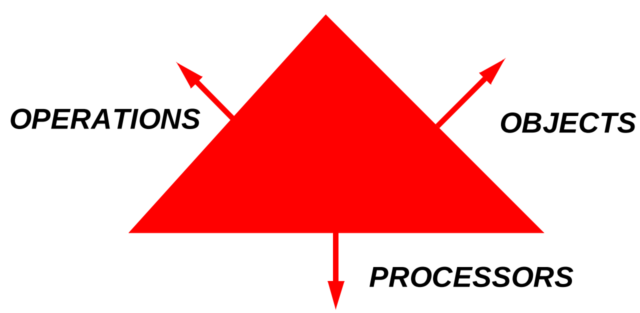
print_diagnostics

prepare_for_maintenance

restart_job

THE BASIC TRIANGLE OF COMPUTATION

Computing consists of applying *operations* to *objects*; to do so requires the appropriate mechanisms – *processors*.



SEPARATE ENTITIES

A call of the form x.f (a) will have a different semantics depending on whether Current and x are handled by the same or different processors.

The semantics must of course be immediately clear from the software text. Need to declare whether client processor is the same as supplier processor or another.

x: separate A

Contrast with the usual

x: A

which guarantees that objects attached to x will be handled by the same processor as the current object.

CONSISTENCY RULE

In the assignment

x := y

if the source y is separate, the target x must be separate too.

Same rule for argument passing.

SEPARATE ENTITIES AND CLASSES

b: separate BOUNDED_QUEUE [SOME_TYPE]

or:

```
separate class BOUNDED_BUFFER [G] inherit

    BOUNDED_QUEUE [G]

end
```

x: BOUNDED_BUFFER [SOME_TYPE]

CREATION

If x is separate, then the creation instruction

create x

grabs a new processor, physical or virtual, and assigns it to handle the object.

Also: it is possible to obtain a separate object as the result of a function. So processors can be allocated outside of Eiffel text proper.

COMMENTS

"Separate" declaration does not specify the processor.

Semantic difference between sequential and concurrent computation narrowed down to difference for separate calls:

- **Precondition semantics**
- **Argument passing semantics**
- **Creation semantics.**

PROCESSOR ASSIGNMENT

The assignment of actual physical resources to (virtual) processors) is entirely dynamic and EXTERNAL to the software text.

Simple notation: Concurrency Control File (CCF)

```
creation

    proc1: sales.microsoft.com (2),
              coffees.whitehouse.gov (5), ...

    proc2: 89.9.200.151 (1), ...
```

Physical resources may be Internet nodes, threads, Unix or Windows processes, etc.

REFERRING TO EXTERNAL OBJECTS

With

```
a: separate SOME_CLASS
```

the value of a at run time is a reference to an object handled by another processor. (Implemented as a proxy object.)

The normal Eiffel clone or deep_clone mechanism would result in inconsistencies (and violates the type constraints).

New mechanism in the Kernel library (ELKS, Eiffel Library Kernel Standard):

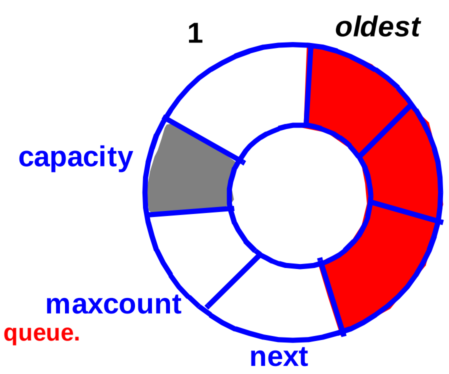
```
b := deep_import (a)
```


PREDEFINED CONSTRUCTS AND LIBRARIES

Define specific details (how many processors...) and scheduling policies through libraries.

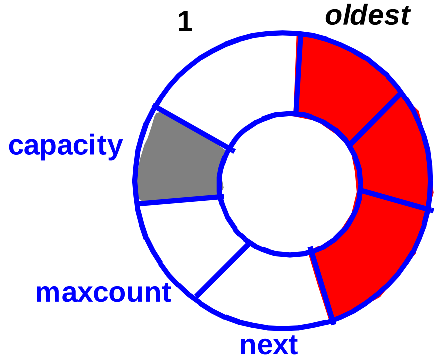
DESIGN BY CONTRACT

```
class BOUNDED_QUEUE [G] feature
    put (x: G) is
            -- Add x to queue.
        require
            not full
        do
            ...
        ensure
            not empty
        end

    remove: G is
            -- Delete oldest element from queue.
        require
            not empty
        do
            ...
        ensure
            not full
        end
```

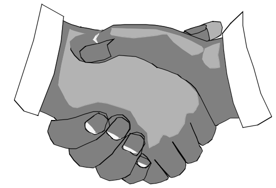


THE CONTRACT MODEL (Continued)

```
item: is
        -- Oldest element.
    require
        not empty
    do
        Result := ...
    end
...
invariant
  maxcount = capacity – 1
  0 <= oldest; oldest <= capacity
  0 <= next; next <= capacity
  abs (next – oldest) < capacity
end
```

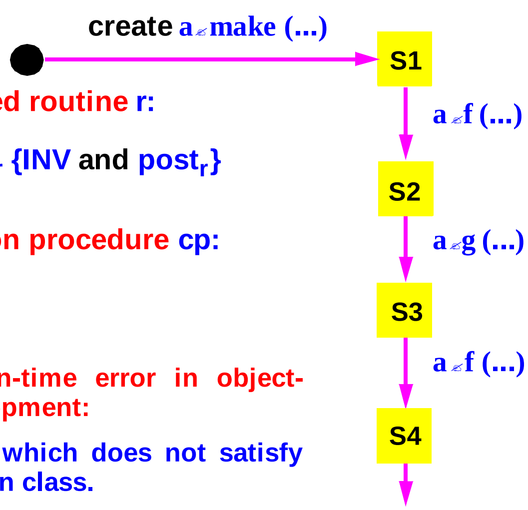


THE CONTRACT OF A FEATURE



put	OBLIGATIONS	BENEFITS
Client	*(Satisfy precondition:)* Make sure queue not full.	*(From postcondition:)* Make queue not empty, *x* added.
Supplier	*(Satisfy postcondition:)* Insert *x*, making sure queue is not empty.	*(From precondition:)* Simpler processing thanks to assumption that queue not full.

THE CORRECTNESS OF A CLASS



(1-n) For every exported routine r:

$\{INV \text{ and } pre_r\}\ body_r\ \{INV \text{ and } post_r\}$

(1-m) For every creation procedure cp:

$\{pre_{cp}\}\ do_{cp}\ \{INV\}$

The worst possible run-time error in object-oriented software development:

- Producing an object which does not satisfy the invariant of its own class.

PROVABILITY

Proof rule for routines:

$$\frac{\{ INV\ ?\ \bigwedge_{p\ ?\ Pre(r)} p \}\ \text{Body}(r)\ \{ INV\ ?\ \bigwedge_{q\ ?\ Post(r)} q \}}{\{ \bigwedge_{p\ ?\ Pre(r)} p' \}\ \text{Call}(r)\ \{ \bigwedge_{q\ ?\ Post(r)} q' \}}$$

In other words: to prove the validity of *all* calls, it suffices to prove (once!) the correctnes of the body

EXPRESS MESSAGES

AND THE UNIT OF GRANULARITY

An express message is a message that must be treated right away, interrupting any current routine call.

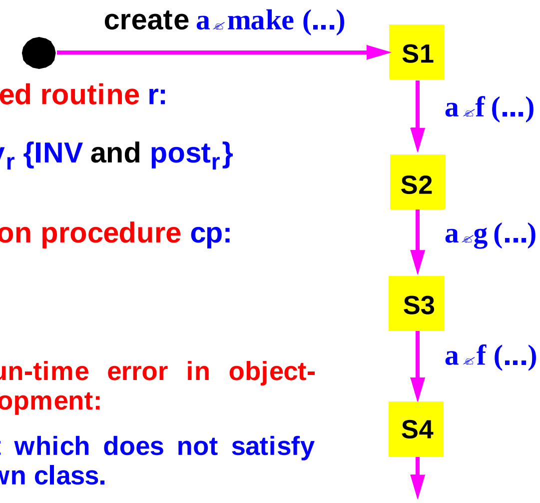
- But: how do we preserve the consistency of objects (invariants)?

The model will support a restricted form of express messages, which does not conflict with provability.

Unit of granularity for mutual exclusion is routine call.

But: can be interrupted, causing an exception.

THE CORRECTNESS OF A CLASS



(1-n) For every exported routine r:

{INV and pre_r} $body_r$ {INV and $post_r$}

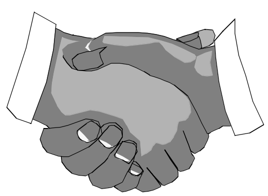
(1-m) For every creation procedure cp:

{pre_{cp}} do_{cp} {INV}

The worst possible run-time error in object-oriented software development:

- Producing an object which does not satisfy the invariant of its own class.

THE CONTRACT OF A FEATURE



put	OBLIGATIONS	BENEFITS
Client	(*Satisfy precondition:*) Make sure queue not full.	(*From postcondition:*) Make queue not empty, *x* added.
Supplier	(*Satisfy postcondition:*) Insert *x*, making sure queue is not empty.	(*From precondition:*) Simpler processing thanks to assumption that queue not full.

WHAT BECOMES OF THE CONTRACT MODEL?

"NO HIDDEN CLAUSES"

```
q: BOUNDED_QUEUE [X]
a: X
...
if not q.full then
    q.put (a)
end
```

Or:

```
q.remove
q.put (x)
```

But: this does not work for separate threads of control!

What do preconditions now mean?

RESERVING AN OBJECT

q: separate BOUNDED_QUEUE [X]; a: X
...
a := q.item

... Other instructions (not calling remove) ...

q.remove

How do we guarantee that item and remove apply to the same buffer element?

RESERVING AN OBJECT (Continued)

Just use encapsulation. Argument passing serves as reservation. If object busy (processor not available), block object; processor will service other object if possible.

RESERVING AN OBJECT (Continued)

With the class as shown on the following page, the call

put (q)

will block until:

- **q is available.**
- **The precondition not q.full is true.**

The new rule only affects:

- **Separate arguments.**
- **Precondition clauses which include calls on separate targets (i.e. x.f with x separate).**

RESERVING AN OBJECT

```
class BUFFER_ACCESS [X] feature

  put (q: separate BOUNDED_QUEUE [G]; x: G) is

        -- Insert x into q, waiting if necessary until there is room.

      require
         not q.full

      do
         q.put (x)

      ensure

         not q.empty

      end
```

RESERVING AN OBJECT (Continued)

```
remove (q: separate BOUNDED_QUEUE [G]) is
        -- Remove an element from q, waiting if necessary
        -- until there is such an element.
    require
        not q.empty
    do
        q.remove
    ensure
        not q.full
    end

item (q: separate BOUNDED_QUEUE [G]): G is
        -- Oldest element not yet consumed
    ... Left to reader ...
end
```


BASIC SEMANTIC RULES

If *a* is separate, a call of the form

p (..., a, ...)

will block the client until the object attached to a is available.

In addition, if p has a precondition including a call of the form

```
require
    ... Other clauses ...
    a.f
```

(again for separate a), then the call will block until the precondition is satisfied.

THE ORIGINAL PROOF RULE

$$\frac{\{INV\ ?\ \bigwedge_{p\ ?\ Pre(r)} p\}\ Body(r)\ \{INV\ ?\ \bigwedge_{q\ ?\ Post(r)} q\}}{\{\bigwedge_{p\ ?\ Pre(r)} p'\}\ Call(r)\ \{\bigwedge_{q\ ?\ Post(r)} q'\}}$$

THE NEW PROOF RULE

$$\frac{\{INV\ ?\ \bigwedge_{p\ ?\ Nonsep\_Pre(r)} p\}\ Body(r)\ \{INV\ ?\ \bigwedge_{q\ ?\ Nonsep\_Post(r)} q\}}{\{\bigwedge_{p\ ?\ Nonsep\_Pre(r)} p'\}\ Call(r)\ \{\bigwedge_{q\ ?\ Nonsep\_Post(r)} q'\}}$$

Nonsep_pre (r): set of clauses in r's precondition which do not involve any separate calls.

Similarly for Nonsep_post (r).

WAIT BY NECESSITY

(SOURCE: DENIS CAROMEL)

```
r (..., t: separate SOME_TYPE, ...) is
    do
        ...
        t.f (...)
        other_instructions
    end
```

When do we wait?

WAIT BY NECESSITY

For example:

```
r (..., t: separate SOME_TYPE, ...) is
    do
        ...
        t.p (...)
        other_instruction_1
        ...
        other_instruction_n
        k := t.some_value        <-- WAIT HERE
    end
```

Wait on queries (calls to attributes and functions), not procedure calls.

BLOCKING SEMANTICS

IS NOT ALWAYS APPROPRIATE

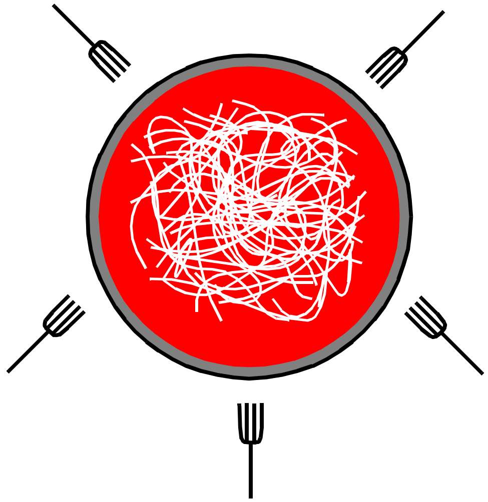
```
f: FILE
...
if f /= Void and then f.readable then

  f.some_input_routine

      -- some_input_routine is any routine that reads
      -- data from the file; its precondition is readable.
end
```


DUELS

Request immediate service: immediate_service
Accept immediate service: yield

Challenger? ? ?Holder	normal_service	immediate_service
insist	Challenger waits	Exception in challenger
yield	Challenger waits	Exception in holder; serve challenger.

DINING PHILOSOPHERS



DINING PHILOSOPHERS

```
separate class PHILOSOPHER creation

      make

inherit

  PROCESS
      rename setup as getup end

feature {BUTLER}

  step is
      do
         think ; eat (left, right)
      end
```



```
feature {NONE}

         -- The two required forks:
  left, right: separate FORK

  getup is
         -- Take any necessary initialization action.
      do ... end

  think is
         -- Any appropriate action.
      do ... end

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


A BINARY TREE CLASS

```
class BINARY_TREE [G] feature

  left, right: BINARY_TREE [G]

  nodes: INTEGER is
         -- Number of nodes in this tree
      do
         Result := node_count (left) + node_count (right) + 1
      end
feature {NONE}
  node_count (b: BINARY_TREE [G]): INTEGER is
         -- Number of nodes in b
      do
         if b /= Void then
            Result := b.nodes
         end
      end
end
```

A BINARY TREE CLASS: PARALLEL VERSION

```
separate class BINARY_TREE [G] feature

  left, right: BINARY_TREE [G]

  ... Other features ...

  nodes: INTEGER

  update_nodes is
        -- Update nodes to reflect number of nodes in this tree.
     do
        nodes := 1
        compute_nodes (left); compute_nodes (right)
        adjust_nodes (left); adjust_nodes (right)
     end
```



```
feature {NONE}
  compute_nodes (b: BINARY_TREE [G]) is
        -- Update information about the number of nodes in b.
     do
        if b /= Void then
           b.update_nodes
        end
     end
  adjust_nodes (b: BINARY_TREE [G]) is
        -- Adjust number of nodes from those in b.
     do
        if b /= Void then
           nodes := nodes + b.nodes
        end
     end
end
```


EXAMPLES IN THE BOOK

Coroutines

Locking a resource — semaphores

An elevator control system

A watchdog mechanism (execute an action, but take control back if not done after *t* seconds).

STATUS

Partial implementation.

- **Unix (SunOS, Solaris, HP etc.).**
- **.NET**
-

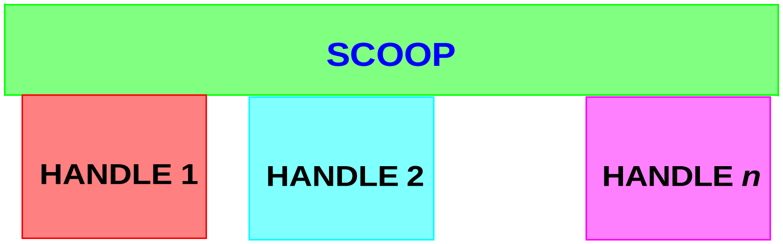
John Potter, UTS

```
hold a until a.some_condition then
   ...
end
```

TWO-LEVEL ARCHITECTURE

As with other Eiffel products (EiffelVision graphical library, EiffelStore relational database library), 2-level architecture:

- **General-purpose top layer (SCOOP).**
- **Several architecture-specific variants at the bottom layer (SCOOP handles).**



Current handle is process-based. Next: multi-threading implementation.

ISSUES AND FUTURE PLANS

Issues:

- **Dual semantics of assertions.**
- **Rule that target of a separate call must be formal argument.**

Hard issues:

- **Deadlock avoidance.**
- **Proof rules and practical proofs (?).**
- **Fairness.**

More work:

- **Various implementations (distributed systems, shared memory, coroutines...).**
- **Processor-CPU association.**