The SCOOP model

Bertrand Meyer
Sequential programming

Used to be messy

Still hard but:

- Structured programming
- Data abstraction & object technology
- Design by Contract
- Genericity, multiple inheritance
- Architectural techniques

Switch from operational reasoning to logical deduction (e.g. invariants)
Concurrent, multithreaded, distributed...

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...
This mechanism

- **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

\[ x \colon CX \]

\[ x.r(a) \]

**Client**

\[ \text{previous\_instruction} \]

\[ x.r(a) \]

\[ \text{next\_instruction} \]

**Supplier (CX)**

\[ r(a \colon A) \text{ is} \]

\[ \text{require} \]

\[ a \neq \text{Void} \]

\[ \text{ensure} \]

\[ \text{not} \ a \text{ is\_empty} \]

\[ \text{end} \]

**Processor**
Object-oriented computation

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

\{Pre_r \textbf{and} \text{INV}\} \text{ body}_r \{Post_r \textbf{and} \text{INV}\} \\
\hline \\
\{Pre'_r\} x.r (a) \{Post'_r\}
Reasoning about objects

Only $n$ proofs if $n$ exported routines!

\[
\{\text{Pre}_r \text{ and INV}\} \quad \text{body}_r \quad \{\text{Post}_r \text{ and INV}\}
\]

\[
\{\text{Pre}_r'\} \ x.r (a) \quad \{\text{Post}_r'\}
\]
In a concurrent context

Only $n$ proofs if $n$ exported routines?

\[
\{\text{Pre}_r \text{ and INV}\} \quad \text{body}_r \quad \{\text{Post}_r \text{ and INV}\}
\]

\[
\hline
\{\text{Pre}_r'\} \times r(a) \quad \{\text{Post}_r'\}
\]

Client 1, $r_1$  
Client 2, $r_2$  
Client 3, $r_3$
Mutual exclusion rule

At most one feature may execute on any one object at any one time
Feature call: sequential

\[
x \cdot r(a)
\]

\[
x: CX
\]

Client

\[
\text{previous\_instruction}
\]

\[
x \cdot r(a)
\]

\[
\text{next\_instruction}
\]

Supplier (\(CX\))

\[
r(a: A) \text{ is}
\]

\[
\text{require}
\]  
\[
a \neq \text{Void}
\]

\[
\text{ensure}
\]  
\[
\text{not } a \text{ is\_empty}
\]

\[
\text{end}
\]

Processor
Feature call: asynchronous

\[ x.r(a) \]

\( x: \text{separate } CX \)

\begin{align*}
\text{Client} & \quad \text{Supplier (}CX\text{)} \\
\text{Client processor} & \quad \text{Supplier processor}
\end{align*}

\begin{align*}
\text{previous_instruction} & \quad r(a, A) \text{ is} \\
\text{next_instruction} & \quad \text{require } a \neq \text{Void} \\
& \quad \text{ensure } \text{not } a. \text{is_empty} \\
& \quad \text{end}
\end{align*}
Separateness rule

Calls to non-separate objects are synchronous

Call to separate objects are asynchronous
Feature call: asynchronous

\[ x \cdot r(a) \]

\( x: \text{separate } CX \)

**Client**

- previous_instruction
- \( x \cdot r(a) \)
- next_instruction

**Supplier (CX)**

- \( r(a, A) \) is
- require \( a \neq \text{Void} \)
- ensure not \( a \) is_empty
- end

Client processor

Supplier processor
Feature call: asynchronous

\[ x.r(a) \]

\( x \): separate \( CX \)

**Client**

- previous_instruction
- \( x.r(a) \)
- next_instruction

**Client processor**

\[ r(a: A) \text{ is}
\require
\text{a /= Void}
\ensure
\text{not}
\text{a.is_empty}
\text{end} \]

**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:

- \( x: CX \)
- \( x: \text{separate} \ CX \)
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
Another consistency rule

In

\[ x := y \]

if \( y \) is separate, \( x \) must be separate too.
If no access control

\[ x \text{. separate } CX \]

\[
\]

\[ x.r(a) \]

\[ y := x.f \]
If no access control

\[
\text{my\_stack: separate STACK [SOME\_TYPE]}
\]

...  

\[
\text{my\_stack.push(a)}
\]

\[
y := \text{my\_stack.top}
\]
Access control policy

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

\[
\text{put (b: separate STACK [T]; value: T) is}
\]

\[
\begin{align*}
\text{do} \\
&& b.push (value) \\
\text{end}
\end{align*}
\]

-- Push value on top of b.
Access control policy

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

\[
\text{store}(b: \text{separate \ buffering}[T]; \text{value: } T) \text{ is }
\]

\[
\quad \text{- Store value into } b.
\]

\[
\begin{align*}
do \\
\text{\hspace{0.5cm}b.put(value)} \\
\end{align*}
\]

end

To use separate object:

\[
\text{my_buffer: separate \ buffering}[\text{INTEGER}]
\]

\[
\text{create my_buffer}
\]

\[
\text{store(my_buffer, 10)}
\]
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
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 (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)
```
From preconditions to wait-conditions

```pascal
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)

If buffer is separate,
```

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

**Client:**

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

**Supplier:**

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

Client:

if **not** my_buffer.is_full

?? ??

then

store (my_buffer, x)

end

Supplier:

\[
\text{store}(b: \text{BUFFER}[T]; \text{value}: T) \text{ is}
\]

-- Store value into b.

require

- \(b.\text{is_full}\) not
- \(value > 0\)

do

- \(b.\text{put}(value)\)

ensure

- \(b.\text{is_empty}\) not

end

...
What happens to preconditions?

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

This becomes the basic synchronization mechanism
A separate precondition causes the client to wait

Separate precondition: \textit{a.condition (...)}
where \textit{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:

\[ x.f \]
\[ x.g(a) \]
\[ y.f \]
\[ \ldots \]
\[ value := x.some_query \]

Wait here!
Resynchronization rule

Clients wait for resynchronization on queries
Interrupts?

Can we snatch shared object from its current holder?

Execute $\text{holder}\cdot r(b)$ where $b$ is separate

Another object executes $\text{challenger}\cdot s(b)$

Normally, $\text{challenger}$ would wait

What if $\text{challenger}$ is impatient?
## The duel mechanism

### Library features

<table>
<thead>
<tr>
<th>Challenger →</th>
<th>normal_service</th>
<th>immediate_service</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ Holder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retain</td>
<td>Challenger waits</td>
<td>Exception in challenger</td>
</tr>
<tr>
<td>yield</td>
<td>Challenger waits</td>
<td>Exception in holder; serve challenger</td>
</tr>
</tbody>
</table>
Extending duels

Timing limits

Priorities (for real-time processing)
Example: class *PROCESS*

defered class

   *PROCESS*

feature -- Status report

   *over: BOOLEAN is*

      -- Must execution terminate now?

defered end

feature -- Basic operations

   *setup is*

      -- Prepare to execute process (default: nothing).

do end

   *step is*

      -- Execute basic process operations.

defered 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

\[\text{buff: BUFFER\_ACCESS[MESSAGE]}\]
\[\text{my\_buffer: BOUNDED\_BUFFER[MESSAGE]}\]

\text{create my\_buffer}
\text{create buff.make (my\_buffer)}

\text{buff.put (my\_buffer, my\_message)}
\ldots
\text{buff.put (my\_buffer, her\_message)}
\ldots
\text{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.
<table>
<thead>
<tr>
<th>Challenger →</th>
<th>normal_service</th>
<th>immediate_service</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>retain</strong></td>
<td>Challenger waits</td>
<td>Exception in challenger</td>
</tr>
<tr>
<td><strong>yield</strong></td>
<td>Challenger waits</td>
<td>Exception in holder; serve challenger</td>
</tr>
</tbody>
</table>
Two-level architecture of SCOOP

Adaptable to many environments
.NET remoting is current platform
Mapping processors to physical resources

Concurrency Control File (CCF)

```plaintext
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:

![Diagram showing class inheritance with separate client and supplier](image)
### SCOOPLI emulation of SCOOP concepts

<table>
<thead>
<tr>
<th><strong>SCOOP</strong></th>
<th><strong>SCOOPLI</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x: \text{separate } X )</td>
<td>( x: \text{SEPARATE}_X )</td>
</tr>
<tr>
<td>( x: X ) -- class ( X ) is separate</td>
<td>-- ( \text{SEPARATE}_X ) inherits from ( X ) and</td>
</tr>
<tr>
<td></td>
<td>-- ( \text{SEPARATE}_\text{SUPPLIER} )</td>
</tr>
<tr>
<td>( r(x, y) )</td>
<td>separate_execute ([( x, y )], \text{agent} ( r(x, y) ),</td>
</tr>
<tr>
<td>-- ( x ) and ( y ) are separate</td>
<td>\text{agent} ( r_\text{precondition} )</td>
</tr>
<tr>
<td>( r(x: \text{separate } X; y: \text{separate } Y) )</td>
<td>( r_\text{precondition}: \text{BOOLEAN} ) is</td>
</tr>
<tr>
<td>is</td>
<td>\text{do}</td>
</tr>
<tr>
<td>\text{require}</td>
<td>\text{Result} := not ( x ).is_empty \text{ and } ( y).count &gt; 5</td>
</tr>
<tr>
<td>not ( x).is_empty \</td>
<td>\text{end}</td>
</tr>
<tr>
<td>( y).count &gt; 5 \</td>
<td>-- client class inherits from</td>
</tr>
<tr>
<td>( i &gt; 0 ) -- ( i ) non-separate</td>
<td>-- class ( \text{SEPARATE}_\text{CLIENT} )</td>
</tr>
<tr>
<td>( x /= \text{Void} )</td>
<td></td>
</tr>
<tr>
<td>\text{do}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-- ...</td>
<td></td>
</tr>
</tbody>
</table>
**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
Dynamic diagram

Scenario: Pressing the cabin button to move the elevator

1 Cabin button calls `elevator.accept (target)`
2 Elevator calls `engine.move (floor)`
3 Engine calls `gui_main_window.move_elevator (cabin_number, floor)`
4 Engine calls `elevator.record_stop (position)`
Class BUTTON

separate class

   BUTTON

feature

   target: INTEGER

end
Class `CABIN_BUTTON`

```plaintext
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
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**

```plaintext
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
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!
Future work & open problems

Other “handles”
Distribution and Web Services
Prevent deadlock, extend access control policy
Extend for real-time
  - Duel mechanism with priorities
  - Timing assertions?

Integrate with Eiffel Software compiler