SCOOP: an introduction

Bertrand Meyer
Chair of Software Engineering, ETH Zurich, Switzerland
http://sei.inf.ethz.ch

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!

The SCOOP model

- Simple Concurrent Object-Oriented Programming
- High-level concurrency mechanism
- Fully uses inheritance and other O-O techniques
- Applicable to many physical setups: multiprocessing, multithreading, distributed execution, Web services...
- Based on Design by Contract and Eiffel concepts

Object-oriented computation

To perform a computation is
- to use certain processors
- to apply certain actions
- to certain objects.

What makes an application concurrent?

Processor:
autonomous thread of control supporting sequential execution of instructions on one or more objects

Can be implemented as:
- Process
- Thread
- AppDomain (.NET)
- Web service

Feature call: synchronous

Processor

Object 1

Object 2

previous_instruction
x.f(a)
next_instruction

f(a: A)
require
a /= Void
ensure
not a.empty

(CLASS_X)

(CLASS_X)
Separate feature call (asynchronous)

\[ x: \text{separate} \text{CLASS}_X \]

\[ x.f(a) \]

Object 1

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

Object 2

\( f(a: A) \)

- require
- \( a \neq \text{Void} \)
- ensure
- \( \text{not a.empty} \)

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

\[
\text{store (buffer: separate BUFFER [INTEGER]; value: INTEGER)}
\]

- Store value into buffer.
  - \( \text{buffer.put (value)} \)
- \( \text{end} \)

Call with separate argument gives exclusive access:

\[
\text{store (my_buffer, 10)}
\]

Contracts in Eiffel

\[
\text{store (buffer: separate BUFFER [INTEGER]; value: INTEGER)}
\]

- Store value into buffer.
  - \( \text{buffer.put (value)} \)
  - \( \text{end} \)
  - \( \text{store (my_buffer, 10)} \)

- If \( b \) is separate, precondition becomes \text{wait condition} (instead of correctness condition)

From preconditions to wait-conditions

\[
\text{store (buffer: separate BUFFER [INTEGER]; value: INTEGER)}
\]

- Store value into buffer.
  - \( \text{buffer.put (value)} \)
  - \( \text{end} \)
  - \( \text{store (my_buffer, 10)} \)

- If buffer is separate,

Wait by necessity

- No special mechanism for client to resynchronize with supplier after separate call.

  - The client will wait only when it needs to:
    - \( x.f \)
    - \( x.g(a) \)
    - \( y.f \)
    - \( \text{value := x.some_query} \)

  - Wait here!

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**

<table>
<thead>
<tr>
<th>Challenger →</th>
<th>normal_service</th>
<th>immediate_service</th>
</tr>
</thead>
<tbody>
<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>

**Mapping processors to physical resources**

Concurrency Control File (CCF)

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

**Two-level architecture of SCOOP**

- Can be implemented on various platforms
- Microsoft .NET is our reference platform

![SCOOPLI: Library for SCOOP](image)

- Library-based solution
- Implemented in Eiffel for .NET
  (from Eiffel Software: EiffelStudio / ENViSIO! 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](image)
**SCOOPSI Architecture**

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

**Example: Dining philosophers**

```plaintext
class PHILOosopher inherit
process
  rename setup as setup
  redefine step end
feature [BUFFER]
  step is
    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

```plaintext
buf: BUFFER_ACCESS [MESSAGE]
my_buffer: BOUNDED_BUFFER [MESSAGE]

create my_buffer
create buf_make (my_buffer)
buf.put (my_buffer, my_message)
...
buf.put (my_buffer, her_message)
...
my_query := buf.item (my_buffer)
```

**Example: elevator**

For maximal concurrency, all objects are separate

**Dynamic diagram**

```
+-----------+    +-----------+    +-----------+
|  CABIN    | ----| ELEVATOR  | ----| ENGINE   |
| BUTTON    |    |           |    |          |
+-----------+    +-----------+    +-----------+
| GUI_MAIN_WINDOW |    |            |    |
+-----------+    +-----------+    +-----------+
```

**Class BUTTON**

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

**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 **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
            request(e)
        end
end
```

Class **ELEVATOR**

```plaintext
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
end
```

Class **ELEVATOR**

```plaintext
feature (NONE) -- Implementation
    process_request is
        -- Handle new pending request, if any.
        local
            pending: QUEUE [INTEGER]
            pending_remove boolean
        if not pending_is_empty then
            pending := pending_item ;
            actual_process(pending, floor)
            pending_remove := true
        end
actual_process(m: MOTOR; floor: INTEGER) is
    -- Handle new pending request, if any.
        do
            moving := true; move(floor)
        end
feature (NONE) -- Implementation
    pulser: MOTOR ; pending: QUEUE [INTEGER]
end
```

Class **MOTOR**

```plaintext
separate class MOTOR feature (ELEVATOR)
    move(floor: INTEGER) is
        -- Go to floor; on arrival, report
        do
            gul_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
                record_stop(position) end
        feature (NONE)
            cabin: ELEVATOR ; position: INTEGER
        end
end
```

**Distributed execution**

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

**Conclusions**

- **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**

- **SCOOPL# library**
  - .SCOOPL#-based syntax
  - Implemented on .NET
  - Distributed execution with .NET Remoting
Current work

- Prevent deadlock
  - Statically checkable rules
  - Run-time deadlock avoidance
- Extend access control policy
  - Multiple locking of separate objects
- Extend for real-time
  - Duel mechanism with priorities
  - Timing assertions?
- Integrate with Eiffel Software compiler
- Direct support for processors in the CLI?