SCOOP: an introduction

Bertrand Meyer
Chair of Software Engineering,
ETH Zürich, Switzerland
http://oe.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

x: CLASS_X
...
x.f(a)

Object 1

previous_instruction
x.f(a)
next_instruction

(CLASS_T)

Processor

Object 2

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

(CLASS_X)
Separate feature call (asynchronous)

\[ x: \text{separate} \text{CLASS}_X \]
\[
\cdots
\]
\[ x.f \,(a) \]

Access control policy

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

\[
\text{store (buffer: separate BUFFER [INTEGER]; value: INTEGER)}
\]
\[
\quad \text{is}
\]
\[
\quad \quad \text{-- Store value into buffer.}
\]
\[
\quad \quad \text{do}
\]
\[
\quad \quad \quad \text{buffer.put (value)}
\]
\[
\quad \quad \quad \text{end}
\]

Call with separate argument gives exclusive access:

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

Contracts in Eiffel

\[
\text{store (buffer BUFFER [INTEGER]; value: INTEGER)}
\]
\[
\quad \text{is}
\]
\[
\quad \quad \text{-- Store value into buffer.}
\]
\[
\quad \quad \text{require}
\]
\[
\quad \quad \quad \text{not buffer.full} \quad \text{Precondition}
\]
\[
\quad \quad \text{do}
\]
\[
\quad \quad \quad \text{buffer.put (value)}
\]
\[
\quad \quad \quad \text{ensure}
\]
\[
\quad \quad \quad \quad \text{not buffer.empty}
\]
\[
\quad \quad \quad \text{end}
\]
\[
\text{store (my_buffer, 10)}
\]

- If \( b \) is separate, precondition becomes \textbf{wait condition} (instead of correctness condition)
From preconditions to wait-conditions

store (buffer: separate BUFFER [INTEGER]; value: INTEGER)
  is
    require
      not buffers_full
    value > 0
  do
    buffer.put (value)
  ensure
    not buffers_empty
  end
  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:
  \[ x.f \]
  \[ x.g(a) \]
  \[ y.f \]
  ...
  \[ value := x.some_query \]

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>1 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>

**Mapping processors to physical resources**

Concurrent Control File (CCF)

```plaintext
create system
  "Lincoln" (4): "c:\prog\appl\appl.exe"
  "noosevelt" (2): "c:\prog\appl\appl2.dll"
  "Current" (5): "c:\prog\appl\appl3.dll"
end
external database_handler: "Jefferson" port 9000
  ATM_handler: "gates" port 8001
end
default
  port: 8000; instance: 10
end
```

**Two-level architecture of SCOOP**

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

![Two-level architecture diagram]
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:

SCOOPLI emulation of SCOOP concepts

<table>
<thead>
<tr>
<th>SCOOP</th>
<th>SCOOPLI</th>
</tr>
</thead>
</table>
| x separate y      | x: SEPARATE_Y
                    |     - SEPARATE_Y \_ handles from x and |
                    |     - SEPARATE_SUPPLIER |
| (x, y)             | separate_process \(x, y\), |
                    |     - agent \(x, y\), |
                    |     - agent _precondition |
| x₁ and y₁ separate | _precondition: BOOLEAN bs |
|                    |     do |
| in merge           |     | if x₁ empty and y₁ empty and | |
|                    |     | count > 5  |
|                    |     | do  |
|                    |     | y₁ = x₁ empty and y₁ empty and  |
|                    |     | count > 5  |
|                    |     | end |
|                    |     | client class \_ handles from |
|                    |     | \- class \_SEPARATE_CLIENT |
• SEPARATE_HANDLER: locking; checking wait conditions; scheduling of requests
• PROCESSOR_HANDLERS: execute separate calls; implement processors
Example: elevator

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

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

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
    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
end

Class ELEVATOR

feature {MOTOR} -- Implementation

process_request is
  -- Handle new pending request, if any.
  local
    pending: QUEUE [INTEGER]
    pending_is_empty: BOOLEAN
  in
    if not pending_is_empty then
      floor := pending.item; actual_process (pull: floor)
      pending.remove
    end

actual_process (m: MOTOR; floor: INTEGER) is
  -- Handle new pending request, if any.
  do
    moving := true
    move (floor)
  end

feature {MOTOR} -- Implementation

pull: MOTOR; pending: QUEUE [INTEGER]
Class MOTOR

```java
separate class MOTOR feature (ELEVATOR)
moved (floor: INTEGER) is
- goto floor, once there, report.
do
    gui_main_window.move_elevator (cabinet_number, floor)
signal_stopped (cabinet)
end
signaled_stopped (e: ELEVATOR) is
- Report that elevator is stopped on floor position.
do
    e.record_stop (position)
end
feature (NONE)
cabinet: ELEVATOR; position: INTEGER -- Current floor level.
doi gui_main_window: GUI_MAIN_WINDOW
end
```

Distributed execution

- Processors (AppDomains) located on different machines
- .NET takes care of the “dirty work”
  - Marshaling
  - 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

- SCOOP-LI library
  - SCOOP-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
  - Dual mechanism with priorities
  - Timing assertions?
- Integrate with Eiffel Software compiler
- Direct support for processors in the CLI?