Object-Oriented Software Construction

Bertrand Meyer
Lecture 24:

Concurrent O-O principles
The goal

- Provide a practical and general foundation for concurrent and distributed programming, ensuring reliability, reusability, availability and extendibility.
The issue

- Everyone wants to benefit from concurrency and distribution
  - Physical (networks, multiprocessors)
  - Logical (processors, threads)

- but applications remain extremely difficult to build, especially to build right.

- SCOOP provides:
  - Generality
  - Ease of programming
  - Robustness (including liveness, ...)
  - Availability
  - Extendibility, reusability
Generality

- Provide a simple, general, easy to use concurrency and distribution mechanism for programming concurrent applications:
  - Internet and Web programming.
  - Web services, peer-to-peer...
  - Client-server applications.
  - Distributed processing.
  - Multi-threading.
  - Multiple processes (Unix, Windows 95, Windows NT).
An example: Multithreading

- Increasingly popular
- Notoriously hard to get right; major mistakes remain in released applications; theoretical pitfalls
- Impedance mismatch with O-O programming concepts
- Synchronization often achieved by low-level mechanisms
“I can’t understand why objects [of O-O languages] are not concurrent in the first place”.

Robin Milner, 1991
Formulating 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?
Basic O-O mechanism

- Feature call (message passing):

\[ x.f(a) \]
Concurrent O-O should be easy!

(But: it’s not!)

- Analogies between objects/classes and processes/process-types:
  - General decentralized structure, independent modules.
  - Encapsulated behavior (a single cycle for a process; any number of routines for a class).
  - Local variables (attributes of a class, variables of a process or process type).
  - 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

defered class PROCESS feature
live is

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?
class PRINT the mechanism

PROCESS
  rename
    over as off_line,
    finalize as stop
end

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

  step is
    -- Execute individual actions of an iteration step.
    do
      start_job
      process_job
      finish_job
    end
A printer mechanism (cont’d)

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

- A call of the form \( x.f(a) \) has different semantics when handled by the same or different processors.
- Need to declare whether client processor is the same as supplier processor or another.

\[ x: \text{separate } A \]

- Contrast with the usual

\[ x: A \]

- which guarantees handling by same processor.
Consistency rule

- In 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

\[ b: \text{separate } \text{BOUNDED\_QUEUE}\ [\text{SOME\_TYPE}] \]

or:

\text{separate class } \text{BOUNDED\_BUFFER}\ [G]
\text{inherit}

\text{BOUNDED\_QUEUE}\ [G]
\text{end}

\[ x: \text{BOUNDED\_BUFFER}\ [\text{SOME\_TYPE}] \]
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.
Generality

- “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.
Two-level architecture

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

![Diagram showing a two-level architecture with SCOOP at the top level and handles at the bottom level.](image)

- Initial handles: .NET remoting; POSIX threads;
Two-level architecture

SCOOP

.NET  THREADS  ...

Chair of Software Engineering

OOSC - Summer Semester 2004
Predefined constructs and libraries

- Define specific details (how many processors...) and scheduling policies through libraries.
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.
End of lecture 24