SCOOP
Simple Concurrent Object-Oriented Programming
Lecture 4 (12):
Advanced OO mechanisms
SCOOP and...

- Inheritance
- Genericity
- Agents
- Priorities
- Real-time
- Deadlocks
Inheritance

- Can we use inheritance as in the sequential world?

- Is multiple inheritance allowed?

- Does SCOOP suffer from inheritance anomalies?
Example: Dining philosophers

```plaintext
class PHILOSOPHER

inherit
  GENERAL_PHILOSOPHER
  PROCESS

rename
  setup as getup

undefine
  getup

end

feature
  step is
    -- Perform a philosopher's tasks.
    do
      think
      eat (left, right)
    end

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

indexing
  description: "The most general notion of process"

defered class PROCESS

feature -- Status report

  over: BOOLEAN is
    -- Should execution terminate now?
    deferred end

feature -- Basic operations

  setup is
    -- Prepare to execute process operations (default: nothing).
    do end

  step is
    -- Execute basic process operations.
    deferred end
Class \textit{PROCESS} (cont.)

\begin{verbatim}
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 -- class \textit{PROCESS}
\end{verbatim}
**Class GENERAL PHILOSOPHER**

```plaintext
class
    GENERAL PHILOSOPHER

create
    make

feature -- Initialization

    make (l, r: separate FORK) is
        -- Define l as left and r as right forks.
        do
            left := l
        end
        right := r

feature {NONE} -- Implementation

    left: separate FORK
    right: separate FORK
```

*Chair of Software Engineering*  Piotr Nienartowski, 14.06.2005
Class GENERAL_PHILOSOPHER (cont.)

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

think is
   -- Any appropriate action or lack thereof
   do
   end

end -- class GENERAL_PHILOSOPHER

class
   FORK
end
Inheritance

- Full support for inheritance (including multiple inheritance)

- Usual rules apply
  - Weakening pre-conditions
  - Strengthening post-conditions

- Most inheritance anomalies eliminated thanks to the proper use of OO mechanisms
  - Thank you, Design by Contract!
Genericity

- Can we use *genericity* as in the sequential world?

- Is *constrained genericity* allowed?

- What is its impact on the *type system*?
Genericity

my_array: ARRAY [X]

my_array: separate ARRAY [X]

my_array: ARRAY [separate X]

my_array: separate ARRAY [separate X]

- Full support for genericity in SCOOP
  - Not fully implemented yet! 😞
Constrained genericity: why

\[my\_array: ARRAY \ [X]\]

\[my\_array.item \ (2) < my\_array.item \ (4)\]

- How do we know that operation “<” is defined for elements of \[my\_array\]?
Constrained genericity: how

- We can declare class ARRAY as
  \textbf{class} ARRAY \([G \rightarrow COMPARABLE]\)

- Class \textit{COMPARABLE} implements operation “<”

- Only subtype of \textit{COMPARABLE} can be actual
generic parameter
Constrained genericity: how

- Only subtype of COMPARABLE can be actual generic parameter:
  
  \[
  \text{my\_array: ARRAY [INTEGER]} \\
  \text{my\_array: ARRAY [STRING]} \\
  \text{but} \\
  \text{my\_array: ARRAY [ANY]} \\
  \text{is invalid}
  \]

- No restrictions on the use of constrained genericity in SCOOP
Genericity: impact on the type system

- If class is declared as

```
class ARRAY [G]
```

then its formal generic parameter is implicitly constrained to `[G -> ANY]`

```
my_array: ARRAY [separate STRING] is invalid
```

because `separate STRING` is not a subtype of `ANY`!
Genericity: impact on the type system

- To allow separate generic parameters, class has to be declared as

  \textbf{class ARRAY [separate } G ]

  (then its formal generic parameter is implicitly constrained to \([G \rightarrow \text{separate } \text{ANY}])

- If we want to constrain the formal generic parameter, we can declare the class as

  \textbf{class ARRAY [G \rightarrow \text{separate } \text{HASHABLE}]}

Chair of Software Engineering
Piotr Nienaltowski, 14.06.2005
Agents

- What are agents?
- Don’t they cause trouble?
- How does SCOOP deal with them?
- Open issue: open arguments
What are agents?

- Agents represent routines ready to be called

  \texttt{agent } x.f \texttt{ is an “object wrapper” for } x.f

- We can use the agent as any other object, e.g. as source of assignment

  \texttt{a: PROCEDURE [ANY, TUPLE]}

  \texttt{a := agent x.f}

- We can also call the agent

  \texttt{a.call \hspace{1cm} this is equal to executing } x.f
Open arguments

- We can fix arguments or leave them open
  -- assume \( g \) is declared as
  -- \( g \) (\( s: \) STRING; \( i: \) INTEGER)
  \( a, b, c, d: \) PROCEDURE [ANY, TUPLE]
  \( a := \) agent \( x.g \) (“Hello world”, 5)
  \( b := \) agent \( x.g \) (\(?\), 5)
  \( c := \) agent \( x.g \) (“Hello world”, \(?\))
  \( d := \) agent \( x.g \) (\(?\), \(?\))

- Open arguments are set at call time
  \( d.call \) ([“Hello world”, 5])
The trouble with agents

\( a: \text{PROCEDURE [ANY, TUPLE]} \)

\[
\text{store (buffer: separate BUFFER [INTEGER]; x: INTEGER)} \text{ is}
\begin{align*}
do
\text{buffer.put (x)}
& a := \text{agent buffer.put (?)} \quad \text{-- Valid!} \\
& \text{fishy_call (x)}
\end{align*}
end
\]

\[
\text{fishy_call (x: INTEGER)} \text{ is}
\begin{align*}
do
& a.call ([x]) \quad \text{-- Valid! But a is a traitor!} \\
end
\end{align*}
\]
Solution: separate agents

a: separate PROCEDURE [ANY, TUPLE]

store (buffer: separate BUFFER [INTEGER]; x: INTEGER) is
  do
    buffer.put (x)
    a := agent buffer.put (?) -- separate agent
    fishy_call (x)
  end

fishy_call (x: INTEGER) is
  do
    a.call ([x]) -- Invalid: a is not a formal argument.
  end
a: separate \textit{PROCEDURE [ANY, TUPLE]}

\textbf{store} \texttt{(buffer: separate BUFFER [INTEGER]; x: INTEGER) is}

\begin{verbatim}
  do
    buffer.put (x)
    a := agent buffer.put (?)
    not_so_fishy_call (a, x)
  end

not_so_fishy_call \texttt{(an_agent: separate PROCEDURE [ANY, TUPLE];
  x: INTEGER) is}

\begin{verbatim}
  do
    an_agent.call ([x])  -- Valid: a is a formal argument.
  end
\end{verbatim}

different semantics for argument passing: lock also \texttt{an_agent.target}
Agents well integrated

Open issue: open arguments
  We cannot check statically that actual arguments conform to formal arguments expected by the agent

Open-target agents are even trickier

\[ a : \text{PROCEDURE} \ [\text{ANY, TUPLE}] \]
\[ a := \text{agent} \ \{X\}.f \]
\[ a.call ([x]) \ -- \text{ equal to } x.f \]
\[ a.call ([y]) \ -- \text{ equal to } y.f \]
Priorities and real-time

- How to satisfy VIP clients?

- The duel mechanism

- How to implement priority scheduling?

- How to provide other real-time capabilities?
SCOOP for real-time systems

- Possibility to execute the request of a VIP client while preempting the current client
  - Duels
  - Priorities

- Timing assertions
  - Maximal (and minimal) execution time
  - Timeouts on actions
  - Periodic and aperiodic activities
Can we snatch a shared object from its current holder?

- *holder* executes 
  \[ \text{holder}.r \ (b) \] 
  where \( b \) is *separate*

- another object *challenger* executes 
  \[ \text{challenger}.s \ (c) \] 
  where \( c \), also *separate*, is attached to the same object as the holder’s \( b \)

- Normally, the *challenger* will wait until the call to \( r \) is over

- What if the *challenger* is impatient?
## Semantics of duels

<table>
<thead>
<tr>
<th>challenger →</th>
<th>normal_service</th>
<th>demand</th>
<th>demand_if_possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ holder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>retain</strong></td>
<td><strong>challenger</strong> waits</td>
<td>exception in <strong>challenger</strong></td>
<td><strong>challenger</strong> waits</td>
</tr>
<tr>
<td><strong>yield</strong></td>
<td><strong>challenger</strong> waits</td>
<td>exception in <strong>Holder</strong></td>
<td>exception in <strong>holder</strong></td>
</tr>
</tbody>
</table>
## Semantics of duels with timeouts

<table>
<thead>
<tr>
<th>challenger →</th>
<th>normal_service</th>
<th>demand_with_timeout (time)</th>
<th>demand_if_possible_with_timeout (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ holder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>retain</td>
<td><strong>challenger</strong> waits</td>
<td>exception in <strong>challenger</strong></td>
<td><strong>challenger</strong> waits</td>
</tr>
<tr>
<td>yield</td>
<td><strong>challenger</strong> waits</td>
<td>exception in <strong>Holder</strong></td>
<td>exception in <strong>holder</strong></td>
</tr>
</tbody>
</table>
## Semantics of duels with timeouts

<table>
<thead>
<tr>
<th>challenger →</th>
<th>normal_service</th>
<th>demand_with_timeout (time)</th>
<th>demand_if_possible_with_timeout (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓ holder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>retain_after_timeout (time)</td>
<td>challenger waits</td>
<td>exception in challenger</td>
<td>challenger waits</td>
</tr>
<tr>
<td>yield_after_timeout (time)</td>
<td>challenger waits</td>
<td>exception in Holder</td>
<td>exception in holder</td>
</tr>
</tbody>
</table>
Extending duels with priorities

Tuning the duel mechanism:

- `holder.set_priority (50)`
- `challenger.set_priority (100)`
- `holder.yield`
- `challenger.demand`

If `challenger.priority > holder.priority` then
  - `holder` will get an exception
  - otherwise `challenger` will get an exception
Priorities

- Extending \textit{yield} to specific clients (e.g. \textit{MOTOR})
  For example \textit{yield}\ (\{\textit{MOTOR}\})

- \textit{set\_priority}\ (\textit{i: INTEGER})
  \textit{yield}\ (\{\textit{MOTOR}\}) can be mapped to priorities

- \textit{yield} could even be mapped to a special case of priority.
Timing in sequential programs

**class**

`TIMING_ASSERTION`

**feature** -- Access

`time_now: TIME`

-- The current time now.

`max_time_duration: TIME_DURATION`

-- Maximal time duration of a feature

`min_time_duration: TIME_DURATION`

-- Minimal time duration of a feature

**end -- class** `TIMING_ASSERTION`
Timing in sequential programs (cont.)

class X
inherit TIMING_ASSERTION
redefine
  default_create
end

feature {NONE} – Creation

  default_create is
do
  create min_time_duration.make_by_fine_seconds (0.05)
  create max_time_duration.make_by_fine_seconds (0.1)
end
Timing in sequential programs (cont.)

feature

   f (a_time_started: TIME) is
    require
       a_time_started_not_void: a_time_started /= void
    do
       ...
       create time_now.make_now
    ensure
       minimal_execution_time:
       (time_now - a_time_started).duration > min_time_duration

       maximal_execution_time:
       (time_now - a_time_started).duration < max_time_duration
    end

end -- class X
• I hate deadlocks, I want to get rid of them!

• Methods
  • Prevention
  • Avoidance
  • Detection
  • Resolution

• The SCOOP way
Deadlock prevention

- Check statically that no deadlocks can ever occur
- Difficult if we do not want to restrict programmers’ choices
- Needs smart compilers
- Example solution (simple but restrictive and not feasible in general)
  - impose total order on all shared resources and require them to be acquired in increasing order and released in decreasing order
Deadlock avoidance

- Check at run-time that giving the task the access to a resource will not introduce any deadlock
- Needs smart scheduler
- Might be costly
Deadlock detection

- Check at run-time whether a deadlock occurred
- Usually amounts to building a task/resource graph (Wait-For-Graph) and checking it for cycles
- Might be very costly, so it is better to avoid it
- Necessary in practice
Deadlock resolution

- Happens at run-time when a deadlock is detected
- Usually done by the scheduler
- One of the tasks involved in the deadlock has to be preempted

Problems
- Choice of the right task to be killed
- How to clean up the mess after that?
The SCOOP way

- **Prevent**
  - Compiler will be smart enough to issue a warning if there is potential deadlock (DbC needs to be extended)
  - Programmer has the choice to avoid the danger, e.g. by modifying the precondition of the concerned routine
  - Formal proof of deadlock freeness would also satisfy the compiler

- **Detect** (since programmer can decide in favour of run-time checks)
  - WFG searched for cycles
  - Daniel Moser’s semester project

- **Resolve**
  - Pick a processor to preempt
  - Duels might help
How does SCOOP fit within the OO?

- Basic mechanism: feature call

- Design by Contract
  - Generalised semantics for preconditions, postconditions, and invariants

- Inheritance
  - No particular restrictions, usual rules apply
  - Most inheritance anomalies eliminated!

- Genericity: full support

- Agents: almost full support
  - Unclear semantics of agents with open target
That’s all, folks!

Questions?