Object-Oriented Software Construction

Bertrand Meyer
Lecture 23:
Agents and event-driven design
Avoiding glue code

Event producer (e.g. GUI)

Direct subscription

Connection object

Business model (application logic)
create source_line_metric.make
("Source_lines",
[
    [feature_scope, agent feature_line_counter],
    [class_scope, agent class_line_counter]
]
)
Error handling example: without agents

\[
\text{action1} \\
\text{if ok1 then} \\
\quad \text{action2} \\
\quad \text{if ok2 then} \\
\quad \quad \text{action3} \\
\quad \quad \ldots \text{ More processing, more nesting } \ldots \\
\quad \text{end} \\
\text{end}
\]
execute ([agent action1, 
agent action2 (...),
agent action3 (...)])

if glitch then
   warning (glitch_message)
end
Normal call vs. agent

- Normal call

  \[ a_0.f(a_1, a_2, a_3) \]

- Agent call (expression): preface it by keyword \texttt{agent}, yielding

  \texttt{agent} \( a_0.f(a_1, a_2, a_3) \)

- For example:

  \[ u := \texttt{agent} \ a_0.f(a_1, a_2, a_3) \]

- This represents the routine, ready to be called. To call it:

  \[ u.call([]) \]
  \[ \text{-- For type of } u, \text{ see next} \]

- Recall original name of agents: “delayed calls”. 
Type of closed agent expression

- In class $C$:

  \[
  f \left( x_1 : T_1 ; x_2 : T_2 ; x_3 : T_3 \right) \text{ is } \\
  \text{do} \\
  \quad \ldots \\
  \text{end} \\
  u := \text{agent } f \left( a_1 , a_2 , a_3 \right)
  \]

- In some other class:

  \[
  a_0 : C \\
  v := \text{agent } a_0.f \left( a_1 , a_2 , a_3 \right)
  \]

- Type of both $u$ and $v$:

  \[
  u , v : \text{PROCEDURE } [ C , \text{TUPLE}] 
  \]
Tuples

- Purposes:
  - Allow manipulation of **sequences of values** with arbitrary number of elements, with simple structure
  - Support “anonymous classes”
  - To allow for function that return **multiple results**
  - Permit type-safe agents
Tuple classes

- Syntax: \textit{TUPLE} \([X, Y, ...]\)
Mathematical model for tuples

- First intuition:
  \( \text{TUPLE} [A, B, C] \) represents the cartesian product \( A \times B \times C \)

- But: \( A \times B \times C \) cannot be mapped to a subset of \( A \times B \)!

- Better model:
  - \( \text{TUPLE} \) represents the set of partial functions from \( \mathbb{N} \) (set of integers) to the set of possible values, whose domain includes the interval \([1 .. n]\) for some \( n \).

- Example of such a function:
  \{<1, "a">, <2, "a">, <3, "a">\}

- An element of \( \text{TUPLE} [A, B, C] \) is a function whose domain includes the interval \([1 .. 3]\))

- So it’s also an element of \( \text{TUPLE} [A, B] \): functions whose domain includes interval \([1 .. 2]\).
Reminder: constrained genericity

- **LIST** \([G]\) (unconstrained): \(G\) represents arbitrary type. May use
  
  \[
  \text{LIST} [\text{INTEGER}]
  
  \text{LIST} [\text{EMPLOYEE}]
  
  \text{LIST} [\text{SHIP}]
  \]

- **SORTABLE_LIST** \([G \rightarrow \text{COMPARABLE}]\)
  (constrained by \text{COMPARABLE})

- \(G\) represents type descending from \text{COMPARABLE}

  \[
  \text{LIST} [\text{INTEGER}]
  
  \text{LIST} [T] \text{ only if } T \text{ descendant of } \text{COMPARABLE}
  \]
Agent types: Kernel library classes

- **ROUTINE***
  
  \[\text{BASE, ARGS} \rightarrow \text{TUPLE}\]

- **PROCEDURE***
  
  \[\text{BASE, ARGS} \rightarrow \text{TUPLE}\]

- **FUNCTION***
  
  \[\text{BASE, ARGS} \rightarrow \text{TUPLE, RES}\]

Inherits from

* Deferred
Features of routine classes

- **call** (values: \(ARGS\))
- **item** (values: \(ARGS\)): \(RESULT\_TYPE\)
  
  -- In FUNCTION only

  ... target, arguments, set_target, set_arguments...

- Introspection features (in progress):
  - **precondition**: FUNCTION \([BASE, ARGUMENTS, BOOLEAN]\)
  - **postcondition**
  - **type**: \(TYPE\)

Features of class \(TYPE\): heirs, parents, routines etc.
Keeping arguments open

- An agent can have both “closed” and “open” arguments.
- Closed arguments set at time of agent definition; open arguments set at time of each call.
- To keep an argument open, just replace it by a question mark:

  \[
  \begin{align*}
  u & := \text{agent } a0.f(a1, a2, a3) \\
  & \quad \text{-- All closed (as before)} \\
  w & := \text{agent } a0.f(a1, a2, ?) \\
  x & := \text{agent } a0.f(a1, ?, a3) \\
  y & := \text{agent } a0.f(a1, ?, ?) \\
  z & := \text{agent } a0.f(?, ?, ?)
  \end{align*}
  \]
Agent types

- Reminder:

\[
\text{PROCEDURE } [\text{BASE, ARGS } \rightarrow \text{TUPLE}]
\]

\[f \ (x_1: T_1; x_2: T_2; x_3: T_3) \text{ is} \]

\[-\text{ in class C}\]

\[\text{do}\]

\[\ldots\]

\[\text{end}\]

\[\text{agent } a_0.f \ (a_1, a_2, a_3) \ \text{PROCEDURE } [\text{C, TUPLE}]\]

\[\text{agent } a_0.f \ (a_1, a_2, \ ?) \ \text{PROCEDURE } [\text{C, TUPLE} \ [\text{T3}]]\]

\[\text{agent } a_0.f \ (a_1, \ ?, a_3) \ \text{PROCEDURE } [\text{C, TUPLE} \ [\text{T2}]]\]

\[\text{agent } a_0.f \ (a_1, \ ?, \ ?) \ \text{PROCEDURE } [\text{C, TUPLE} \ [\text{T2, T3}]]\]

\[\text{agent } a_0.f \ (\ ?, \ ?, \ ?) \ \text{PROCEDURE } [\text{C, TUPLE} \ [\text{T1, T2, T3}]]\]
Calling an agent

\[ a0: C; a1: T1; a2: T2; a3: T3 \]

\[ u := \text{agent } a0.f(a1, a2, a3) \quad \text{PROCEDURE } [C, \text{TUPLE}] \]
\[
\text{PROCEDURE } [C, \text{TUPLE}] \]
\[ u\text{.call ([]} \]
\[\]

\[ v := \text{agent } a0.f(a1, a2, ?) \quad \text{PROCEDURE } [C, \text{TUPLE } [T3]] \]
\[
\text{PROCEDURE } [C, \text{TUPLE } [T3]] \]
\[ v\text{.call ([}a3]) \]

\[ w := \text{agent } a0.f(a1, ?, a3) \quad \text{PROCEDURE } [C, \text{TUPLE } [T2]] \]
\[
\text{PROCEDURE } [C, \text{TUPLE } [T2]] \]
\[ w\text{.call (}a2]) \]

\[ x := \text{agent } a0.f(a1, ?, ?) \quad \text{PROCEDURE } [C, \text{TUPLE } [T2, T3]] \]
\[
\text{PROCEDURE } [C, \text{TUPLE } [T2, T3]] \]
\[ x\text{.call (}a2, a3]) \]

\[ y := \text{agent } a0.f(?, ?, ?) \quad \text{PROCEDURE } [C, \text{TUPLE } [T1, T2, T3]] \]
\[
\text{PROCEDURE } [C, \text{TUPLE } [T1, T2, T3]] \]
\[ y\text{.call (}a1, a2, a3]) \]
Keeping the target open

\[ r := \text{agent} \{T_0\}.f(a_1, a_2, a_3) \]
-- Target open, arguments closed

Type is: \textit{PROCEDURE} \([T_0, \text{TUPLE} [T_0]]\)

Example call: \(r.call([a_0])\)

\[ s := \text{agent} \{T_0\}.f(?, ?, ?) \]
-- Open on all operands
-- Can also be written as just:
\[ \text{agent} \{T_0\}.f \]

Type is: \textit{PROCEDURE} \([T_0, \text{TUPLE} [T_0, T_1, T_2, T_3]]\)

Example call: \(s.call([a_0, a_1, a_2, a_3])\)
Calling an agent: integration example

\[ \int_{a}^{b} \text{my\_function} (x) \, dx \]

\[ \int_{a}^{b} \text{your\_function} (x, u, v) \, dx \]

\text{my\_integrator\_integral} (\text{agent my\_function}, a, b)

\text{my\_integrator\_integral} (\text{agent your\_function (}, u, v), a, b)
The integral function

\[
\text{integral} \\
(f: \text{FUNCTION} [\text{ANY, TUPLE [REAL, REAL]}]; \text{low, high: REAL}): \text{REAL} \text{ is} \\
\text{-- Integral of } f \text{ over the interval } [\text{low, high}]
\]

local
\[
x: \text{REAL} \\
i: \text{INTEGER} \quad \text{do}
\]
from
\[
x := \text{low}
\]
until
\[
x > \text{high}
\]
loop
\[
\text{Result} := \text{Result} + f.\text{item}([x]) \times \text{step}
\]
i := i + 1
x := low + i \times \text{step}
end
end
Calling an agent: iterator

\[
\text{all\_positive} := \\
\text{my\_integer\_list} \cdot \text{for\_all} (\text{agent is\_positive} (\_))
\]

\[
\text{all\_married} := \\
\text{my\_employee\_list} \cdot \text{for\_all} (\text{agent } \{\text{EMPLOYEE}\}.\text{is\_married})
\]
- In class \textit{LINEAR [G]}, ancestor to all classes representing lists, sequences etc.

\begin{verbatim}
for_all (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]) is
  -- Is there no item in structure that doesn't satisfy test?
  do
    from
      start
      Result := True
    until
      off or not Result
    loop
      Result := test.item ([item])
    forth
  end
end
\end{verbatim}
Iterators (cont’d)

for_all
there_exists
do_all
do_if
do_while
do_until
Command classes

- Undo-redo design pattern
- Support for undo, redo
- Class `COMMAND` provides a procedure `execute` and its heir `UNDOABLE_COMMAND` adds `undo`.
- Write a new class for every kind of undoable command.
Command classes (cont’d)

- Traditionally: one command class (descendant of COMMAND) for every kind of command.

- Now, can transform any procedure into command:

  \[
  \text{operation: } \text{PROCEDURE } [\text{CONTEXT}, \text{TUPLE}]
  \]
  \[
  \quad \quad \quad \quad \quad \quad \quad \text{-- Operation to be applied by current command}
  \]

  \[
  \text{make (p: like operation) is}
  \]
  \[
  \quad \quad \quad \quad \quad \quad \quad \text{-- Make p the current command’s operation.}
  \]

  \[
  \text{require}
  \]
  \[
  \quad \quad \quad \quad p_{\text{not void}}: p \neq \text{Void}
  \]

  \[
  \text{do}
  \]
  \[
  \quad \quad \quad \quad \text{operation} := p
  \]

  \[
  \text{ensure}
  \]
  \[
  \quad \quad \quad \quad \text{operation_set: operation} = p
  \]

end
Inline agents (under discussion)

my_employee_list.do_all (do
    count := count + 1
    end)
Lessons: On language design & evolution

- Maximize Signal
  Noise
- Provide one good way to do anything
- Dogmatism / Consistency?
“There is one and only one kind of acceptable language extension: the one that dawns on you with the sudden self-evidence of morning mist. It must provide a complete solution to a real problem, but usually that is not enough: almost all good extensions solve several potential problems at once, through a simple addition. It must be simple, elegant, explainable to any competent user of the language in a minute or two. It must fit perfectly within the spirit and letter of the rest of the language. It must not have any dark sides or raise any unanswerable questions. And because software engineering is engineering, you must see the implementation technique.”
Is this a new programming paradigm?

- Lack of a methodology
End of lecture 23