Trusted Components
Reuse, Contracts and Patterns

Prof. Dr. Bertrand Meyer
Dr. Karine Arnout
Lecture 10: Agents
Agenda for today

- Scope and applications
- The mechanism
- How .NET does it
Scope

- Starting from an object-oriented basis, add a new kind of objects representing potential computations.

- Such objects are called “agents”.

- Earlier names:
  - Delayed calls
  - Routine objects

- Similar to:
  - Delegates (.NET: C#, Visual Basic .NET...)
  - Blocks (Smalltalk)
  - Lambda expressions
Integration example

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

`my_integrator.integral (agent my_function, a, b)`
Agenda for today

- Scope and applications
- The mechanism
- How .NET does it
Open and closed arguments

- **Closed**: set at the time of the agent’s definition
- **Open**: set at the time of any call to the agent
Using a routine from another class

agent some_object.some_routine (?, u, v)

Target
Consider

\[ \text{my\_integer\_list: LIST [INTEGER]} \]

in a class \( C \) that has the function

\[ \text{is\_positive (x: INTEGER): BOOLEAN is} \]
\[ \text{-- Is x positive?} \]
\[ \text{do} \]
\[ \text{Result := (x > 0)} \]
\[ \text{end} \]

To test that all integers in a list are positive:

\[ \text{all\_positive := my\_integer\_list\_for\_all (agent is\_positive)} \]
Consider

\[ \text{my\_employee\_list: LIST [EMPLOYEE]} \]

where class \textit{EMPLOYEE} has the feature

\[ \text{is\_married: BOOLEAN} \]

-- Is this employee married?

To test that all employees in a list are married:

\[ \text{all\_married := my\_employee\_list}\text{.for\_all (agent {EMPLOYEE}.is\_married)} \]
In class $C$:

- $is\_positive\ (x: INTEGER): BOOLEAN$ is...

```
use_agents

local

my_integer_list: LIST [INTEGER]
my_employee_list: LIST [EMPLOYEE]

\[
\text{do}
\text{my_integer_list}\_.\text{for_all} (~agent\ \text{is\_positive (~?~)})
\text{my_employee_list}\_.\text{for_all} (~agent\ \{EMPLOYEE\}\_.\text{is\_married})
\text{end}
\]```

In class $EMPLOYEE$:

- $is\_married: BOOLEAN$ is...

```
\]```
Normal call vs. agent call

- Normal call
  \[ a0.f (a1, a2, a3) \]

- Agent call (expression): preface it by keyword `agent`, yielding
  \[ \text{agent } a0.f (a1, a2, a3) \]

- For example:
  \[ u := \text{agent } a0.f (a1, a2, a3) \]

- This represents the routine, ready to be called. To call it:
  \[ u.call ([]) \]

- Recall original name of agents: “delayed calls”.

-- For type of \( u \), see next
Behind agents: 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: \texttt{TUPLE [X, Y, ...]}

\begin{center}
\begin{tikzpicture}
  \node (a) {TUPLE};
  \node (b) [below of=a] {TUPLE [A]};
  \node (c) [below of=b] {TUPLE [A, B]};
  \draw[->, red] (a) -- (b);
  \draw[->, red] (b) -- (c);
\end{tikzpicture}
\end{center}
Mathematical model for tuples

- First intuition:
  \[ TUPLE \ [ A, B, C ] \] 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:
  - \( TUPLE \) 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:
  \{\langle 1, "a" \rangle, \langle 2, "a" \rangle, \langle 3, "a" \rangle\}

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

- So it’s also an element of \( TUPLE \ [ A, B ] \): functions whose domain includes interval \([1 .. 2]\).
Agent types: Kernel library classes

ROUTINE
[BASE, ARGS -> TUPLE]

PROCEDURE
[BASE, ARGS -> TUPLE]

FUNCTION
[BASE, ARGS -> TUPLE, RES]

* Inherits from * Deferred

item

call
deferred class
ROUTINE [BASE_TYPE, OPEN_ARGS -> TUPLE create default_create end]
...
feature -- Basic operations
call (args: OPEN_ARGS) is
  -- Call routine with operands `args'.
  require
    valid_operands: valid_operands (args)
callable: callable
do
  ...
end
...
feature -- Status report
valid_operands (args: OPEN_ARGS): BOOLEAN is
  -- Are `args' valid operands for this routine?
do
  ...
end
end
Creating vs. calling an agent

- Writing:
  
  \[ \textbf{agent} \textit{my\_feature} \]

  creates an agent, i.e. an object of type \textit{ROUTINE}.

- To call an agent, one needs to execute \textit{call} (with the proper arguments) to this \textit{ROUTINE} object, e.g:

  \[ \textit{my\_routine\_call ([args])} \]
**Type of closed agent expression**

- In class $C$:
  
  $$f (x_1: T_1; x_2: T_2; x_3: T_3) \textbf{is}$$
  
  $\textbf{do}$
  
  $\ldots$
  
  $\textbf{end}$

  $$u := \textbf{agent} f (a_1, a_2, a_3)$$

- In some other class:
  
  $$a_0: C$$

  $$v := \textbf{agent} a_0.f (a_1, a_2, a_3)$$

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

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

  \[
  \textbf{class} \quad \text{PROCEDURE} \left[ \text{BASE, ARGS} \rightarrow \text{TUPLE} \right] \\
  f \left( x_1: T_1; x_2: T_2; x_3: T_3 \right) \textbf{is} \\
  \quad \text{-- In class } C \\
  \quad \text{do} \\
  \quad \quad \ldots \\
  \quad \text{end} \\
  \textbf{agent} \quad a_0.f \left( a_1, a_2, a_3 \right) \quad \text{PROCEDURE} \left[ C, \text{TUPLE} \right] \\
  \textbf{agent} \quad a_0.f \left( a_1, a_2, ? \right) \quad \text{PROCEDURE} \left[ C, \text{TUPLE} \left[ T_3 \right] \right] \\
  \textbf{agent} \quad a_0.f \left( a_1, ?, a_3 \right) \quad \text{PROCEDURE} \left[ C, \text{TUPLE} \left[ T_2 \right] \right] \\
  \textbf{agent} \quad a_0.f \left( a_1, ?, ? \right) \quad \text{PROCEDURE} \left[ C, \text{TUPLE} \left[ T_2, T_3 \right] \right] \\
  \textbf{agent} \quad a_0.f \left( ?, ?, ? \right) \quad \text{PROCEDURE} \left[ C, \text{TUPLE} \left[ T_1, T_2, T_3 \right] \right]
Calling an agent

\[ a_0: C; a_1: T_1; a_2: T_2; a_3: T_3 \]

\[ u := \text{agent } a_0.f(a_1, a_2, a_3) \quad \text{PROCEDURE } [C, \text{TUPLE}] \]
\[ u.call([]) \]

\[ v := \text{agent } a_0.f(a_1, a_2, ?) \quad \text{PROCEDURE } [C, \text{TUPLE} [T_3]] \]
\[ v.call([a_3]) \]

\[ w := \text{agent } a_0.f(a_1, ?, a_3) \quad \text{PROCEDURE } [C, \text{TUPLE} [T_2]] \]
\[ w.call([a_2]) \]

\[ x := \text{agent } a_0.f(a_1, ?, ?) \quad \text{PROCEDURE } [C, \text{TUPLE} [T_2, T_3]] \]
\[ x.call([a_2, a_3]) \]

\[ y := \text{agent } a_0.f(?, ?, ?) \quad \text{PROCEDURE } [C, \text{TUPLE} [T_1, T_2, T_3]] \]
\[ y.call([a_1, a_2, a_3]) \]
Keeping the target open

\[ r := \text{agent} \{T0\}.f (a1, a2, a3) \]
--- Target open, arguments closed

Type is: \textit{PROCEDURE [T0, TUPLE [T0]]}

Example call: \textit{r.call ([a0])}

\[ s := \text{agent} \{T0\}.f (?, ?, ?) \]
--- Open on all operands
--- Can also be written as just: \text{agent} \{T0\}.f

Type is: \textit{PROCEDURE [T0, TUPLE [T0, T1, T2, T3]]}

Example call: \textit{s.call ([a0, a1, a2, a3])}
Calling an agent: Integration example

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

\[ my\_integratorintegral (agent my\_function, a, b) \]

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

\[ my\_integratorintegral (agent your\_function (?, u, v), a, b) \]
The integral function

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

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

all_positive := my_integer_list.for_all (agent is_positive (?))

all_married :=
  my_employee_list.for_all (agent {EMPLOYEE}.is_married)
Iterator features

for_all
do_all
there_exists
do_if
do_while
do_until
for_all vs. do_all

class

LINEAR [G]

feature -- Iteration

for_all (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]): BOOLEAN
-- Is `test' true for all items?

do_all (action: PROCEDURE [ANY, TUPLE [G]])
-- Apply `action' to every item.
-- Semantics not guaranteed if `action' changes the
-- structure; in such a case, apply iterator to clone of
-- structure instead.

Typically used in assertions

Typically used to perform the same action
on all items of a list.

e.g. LINKED_LIST [G] inherits from LINEAR [G]

...
Performance of agents

- One million calls to a routine that does nothing:
  - Directly: 2s (2µs per call)
  - With agents: 14s (14µs per call)

- One million calls to a routine that executes `do_nothing` twenty times:
  - Directly: 33s (33µs per call)
  - With agents: 46s (46µs per call)

- In real applications, no more than 5% of the time spent in feature calls will be calls to agents
  \(\Rightarrow\) Application with agents \(\approx 0.07\) times as slow
  \(\Rightarrow\) Acceptable performance overhead in most cases
Agenda for today

- Scope and applications
- The mechanism
- How .NET does it
**.NET event-delegate mechanism**

- Publisher or subscriber:

  **D1** Introduce descendant `ClickArgs` of `EventArgs` repeating types of arguments of `myProcedure`. (Adds a class.)

  ```csharp
  public class ClickArgs
  {
    int x, y;...
  }
  ```

  **D2** Declare delegate type `ClickDelegate` based on that class. (Adds a type.)

  ```csharp
  public void delegate ClickDelegate
    (Object sender, ClickArgs e);
  ```
D3  Declare new event type **Click** based on the type **ClickDelegate**. (Adds a type.)

```
public event ClickDelegate Click;
```

D4  Write procedure **OnClick** to wrap handling. (Adds a routine.)

```
protected void OnClick (ClickArgs e)
{
    if (Click != null)
        Click (this, e);
}
```

D5  For every event occurrence, create instance of **ClickArgs**, passing arg values to constructor. (Adds a run-time object.)

```
ClickArgs myClickArgs = new ClickArgs (h, v);
```

D6  For every occurrence, trigger event

```
OnClick (myClickArgs);
```
To subscribe a routine `myProcedure`:

D7 ▪ Declare a delegate `myDelegate` of type `ClickDelegate`. (Can be combined with following step as shown next.)

D8 ▪ Instantiate it with `myProcedure` as constructor’s argument.

```
ClickDelegate myDelegate = new ClickDelegate(myProcedure)
```

D9 ▪ Add it to the delegate list for the event.

```
yourButton.Click += myDelegate
```
.NET delegates vs. agents

- **event** is a keyword of the language (special features of a class). But event types should be treated as ordinary objects.

- Cannot have closed arguments: for equivalent of
  
  \[ r(a, ?, ?, b) \]

  must write routine wrapper to be used for delegate.

- Cannot have open target: for equivalent of
  
  \[ \{TYPE\}.r(...) \]

  must write routine wrapper.
Complementary material

- About agents:
  - Eiffel: The Language, 3rd edition (draft), chapter 25
    (User name: Talkitover; password: etl3)

- About .NET delegates:
  - [http://www.inf.ethz.ch/~meyer/ongoing/events.pdf](http://www.inf.ethz.ch/~meyer/ongoing/events.pdf)
    (“5. The .NET event-delegate model”, p 19-30)
End of lecture 10