Introduction to Programming
Bertrand Meyer

Lecture 19: Event-driven programming with agents
Andreas Leitner
Last revised 13 January 2006

Event-driven programming: an example

Specify that when a user clicks this button the system must execute

\[ \text{your\_procedure}(x, y) \]

where \( x \) and \( y \) are the mouse coordinates and \( \text{your\_procedure} \) is a specific procedure of your system.

Handling traditional input

Program drives input:

```pascal
from
read_next_character
until last_character = Enter
i := i + 1
Result.put(last_character, i)
read_next_character
end
```

Avoiding glue code

Event producer (e.g. GUI)

- Direct subscription
- Connection objects

Business model (application logic)

Model View Controller (MVC) Design Pattern

Handling input with modern GUIs

User drives program:

"When a user presses this button, execute that action from my program"

Event-driven programming: an example

Specify that when a user clicks this button the system must execute

\[ \text{your\_procedure}(x, y) \]

where \( x \) and \( y \) are the mouse coordinates and \( \text{your\_procedure} \) is a specific procedure of your system.
Design patterns

- A design pattern is an architectural scheme — a certain organization of classes and features — that provides applications with a standardized solution to a common problem.

- Since 1994, various books have catalogued important patterns. Best known are by Gamma et al. and by Pree.
Triggering an event

- Ask all observers to react to current event.
- Event: `subscribe`, `unsubscribe`, `update`
- Each descendant of `Observer` defines its own version of `update`.

Observer pattern: some limitations

- Each publisher object knows about its observers.
- Only one update procedure in `Observer`.
  - Subscribe to at most one publisher.
  - At most one operation.
- Not reusable — must be coded anew for each application.
- (This is the difference between patterns and components.)

In EiffelVision

```
OK_button.click.action_list.extend
(agent your_procedure)
```

Another approach: event-action table

- More precisely: Event_type - Action Table.
- More precisely: Event_type - Context - Action Table.

<table>
<thead>
<tr>
<th>Event type</th>
<th>Context</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left_click</td>
<td>Yes_button</td>
<td>Save_file</td>
</tr>
<tr>
<td>Left_click</td>
<td>Cancel_button</td>
<td>Reset</td>
</tr>
<tr>
<td>Left_click</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>Right_click</td>
<td>…</td>
<td>Display_Menu</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Mechanisms in other languages

- C and C++: "function pointers".
- C#: delegates (more limited form of agents).
Language note

In non-O-O languages, e.g. C and Matlab, there is no notion of agent, but you can pass a routine as argument to another routine, as in

\[ \int a \times f(a) \, da \]

where \( f \) is the function to integrate. \( \int f(C \) notation, one among many possible ones) is a way to refer to the function \( f \). (We need some such syntax because just \( f \) could be a function call.)

Agents (or delegates in C#) provide a higher-level, more abstract and safer technique by wrapping the routine into an object with all the associated properties.

With .NET delegates: subscriber

1. Declare a delegate \( myDelegate \) of type \( ClickDelegate \). (Usually combined with following step.)
2. Instantiate it with \( yourProcedure \) as argument:
   \[ myDelegate = new ClickDelegate(yourProcedure); \]
3. Add it to the delegate list for the event:
   \[ YES_button.Click += myDelegate; \]

With .NET delegates: publisher (1)

1. Introduce new class \( ClickArgs \) inheriting from \( EventArgs \), repeating arguments types of \( yourProcedure \):
   \[ public class ClickArgs (...) \]
2. Introduce new type \( ClickDelegate \) (delegate type) based on that class
   \[ public void delegate ClickDelegate (Object sender, \( e \)); \]
3. Declare new type \( Click \) (event type) based on the type \( ClickDelegate \):
   \[ public event ClickDelegate Click; \]

With .NET delegates: publisher (2)

4. Write new procedure \( OnClick \) to wrap handling:
   \[ protected void OnClick (ClickArgs \( c \)) \]
   \[ if (\( C != null \)) Click (this, \( c \)); \]
5. For every event occurrence, create new object (instance of \( ClickArgs \)), passing arguments to constructor:
   \[ ClickArgs yourClickargs = new ClickArgs (h, v); \]
6. For every event occurrence, trigger event:
   \[ OnClick (yourClickargs); \]

Using the Eiffel approach (Event Library)

- Event: each event type will be an object
  Example: mouse clicks
- Context: an object, usually representing a user interface element
  Example: a particular button
- Action: an agent representing a routine
  Example: \( your\_procedure \)

Action-event table

Set of triples

[Event, Context, Action]

- Event: any occurrence we track
  Example: a mouse click
- Context: object for which the event is interesting
  Example: a particular button
- Action: what we want to do when the event occurs in the context
  Example: save the file

Action-event table may be implemented as e.g. a hash table.
25 Using the Event Library

The basic class is EVENT_TYPE
On the publisher side, e.g. GUI library:

- (Once) declare event type:
  `click: EVENT_TYPE [TUPLE [INTEGER, INTEGER]]`
- (Once) create event type object:
  `create click`
- To trigger one occurrence of the event:
  `click.publish ([x_coordinate, y_coordinate])`
On the subscriber side, e.g. an application:
  `click.subscribe (agent my_procedure)`

26 Accessing tuple elements

If `t` is a tuple `[a, b, c, ...]`, use

- `t.item(i)` to obtain `i`-th element
  May need assignment attempt:
  ```
  x := t.item(i)
  ```
- `t.put(x, i)` to change `i`-th element

27 Subscriber variants

- `click.subscribe (agent your_procedure)`
- `my_button.click.subscribe (agent your_procedure)`
- `click.subscribe (agent my_procedure (a, ?, ?, b, ?))`
- `click.subscribe (agent other_object, other_procedure)`

28 Tuple type inheritance

```
TUPLE
  TUPLE [A]
  TUPLE [A, B]
  TUPLE [A, B, C]
...
```

A tuple of type `TUPLE [A, B, C]` is a sequence of at least three values, first of type `A`, second of type `B`, third of type `C`.

Tuple values: e.g. `[a1, b1, c1]`

29 Using the Eiffel Event Library

The basic class is EVENT_TYPE
On the publisher side, e.g. GUI library:

- (Once) declare event type:
  `click: EVENT_TYPE [TUPLE [INTEGER, INTEGER]]`
- (Once) create event type object:
  `create click`
- To trigger one occurrence of the event:
  `click.publish ([x_coordinate, y_coordinate])`
On the subscriber side, e.g. an application:
  `click.subscribe (agent my_procedure)`
What you can do with an agent \( a \)

Call the associated routine through the feature "call", whose argument is a single tuple:

\[
a.\text{call}([\text{horizontal\_position}, \text{vertical\_position}])
\]

A tuple

If \( a \) is associated with a function, \( a.\text{item}([\ldots, \ldots]) \) gives the result of applying the function.

Another example of using agents

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

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

\[\text{my\_integrator}\text{.integral}(\text{agent my\_function}, a, b)\]

\[\text{my\_integrator}\text{.integral}(\text{agent your\_function}(?, u, v), a, b)\]

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:

\[
u := \text{agent} a0.f(a1, a2, a3) \quad \text{-- All closed (as before)}
\]

\[
w := \text{agent} a0.f(a1, a2, ?) \quad \text{-- All open}
\]

\[
x := \text{agent} a0.f(a1, ?, a3)
\]

\[
y := \text{agent} a0.f(a1, ?, ?)
\]

\[
z := \text{agent} a0.f(?, ?, ?)
\]

The integration function

\[
\text{integral}( f; \text{FUNCTION}\{\text{ANY, TUPLE}\{\text{REAL}\}, \text{REAL}\}; \text{low, high}; \text{REAL}; \text{REAL} \}
\]

\[
\text{-- Integral of } f \text{ over the interval } [\text{low, high}]
\]

\[
x \in \text{REAL} \quad \text{O \ INTEGER}
\]

\[
\text{do}
\]

\[
\text{from } x := \text{low} \text{ until } x > \text{high} \text{ loop}
\]

\[
\text{Result} := \text{Result} + f.\text{item}(\{x\}) \cdot \text{step}
\]

\[
i := i + 1
\]

\[
x := a * i + \text{step}
\]

\[
\text{end}
\]

Calling the agent

\[
f(x1, T1; x2, T2; x3, T3)
\]

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

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

\[u.\text{call}()\]

\[
v := \text{agent} a0.f(a1, a2, ?)
\]

\[v.\text{call}(a3)\]

\[
w := \text{agent} a0.f(a1, ?, a3)
\]

\[w.\text{call}(a2)\]

\[
x := \text{agent} a0.f(a1, ?,?)
\]

\[x.\text{call}(a2, a3)\]

\[
y := \text{agent} a0.f(?, ?,?)
\]

\[y.\text{call}(a1, a2, a3)\]

Another application: using an iterator

\[
\text{all\_positive} := \text{my\_integer\_list}\text{.for\_all}(\text{agent is\_positive(?)}))
\]
Iterators

In class `LINEAR[G]`, ancestor to all classes for lists, sequences etc., you will find:

```
for_all
there_exists
do_all
do_if
do_while
do_until
```

Declaring an agent

```
p: PROCEDURE[ANY, TUPLE]
  -- Agent representing a procedure,
  -- 0 open arguments

q: PROCEDURE[ANY, TUPLE[X, Y, Z]]
  -- Agent representing a procedure,
  -- 3 open arguments

f: FUNCTION[ANY, TUPLE[X, Y, Z], RES]
  -- Agent representing a procedure,
  -- 3 open arguments, result of type RES
```

Applications of agents

- Undo-redo
- Iteration
- High-level contracts
- Numerical programming
- Introspection (finding out properties of the program itself)

Calling the agent

```
f(a1, a2, a3)
a0, a1, a2, a3, a4: T3
```

```
u := agent a0.f(a1, a2, a3)
```

```
v := agent a0.f(a1, a2, ?)
```

```
w := agent a0.f(a1, ?, a2)
```

```
x := agent a0.f(a1, ?, ?)
```

```
y := agent a0.f(? , ?)
```

EiffelBase classes representing agents

```
call
ROUTINE
defered
+ effective

PROCEDURE
+ FUNCTION
```

End of lecture 20