Introduction to Programming

Bertrand Meyer

Last revised: 2004-01-21
Lecture 21: Event-driven programming and agents
Handling traditional input

Program drives input:

```
from
    read_next_character
until last_character = Enter loop
    i := i + 1
    Result.put (last_character, i)
read_next_character
end
```
User drives program:

“When a user presses this button, execute that action from my program”
Event-driven programming

**PUBLISHERS**
- trigger events

**EVENTS**

**SUBSCRIBERS**
- handle events

- ROUTINE
- ROUTINE
- ROUTINE
A first solution: The observer pattern

** update\(^*\) 

** OBSERVER

\[\text{attach} \quad \text{detach}\]

** PUBLISHER

\[\text{trigger}\]

** update\(^+\) 

** APPCLASS

** LIBCLASS

\[\text{Inherits from}\]

\[\text{Client}\]

\[\text{Deferred}\]

\[\text{Effective}\]
Observer pattern

Publisher keeps a list of observers:

\[\text{subscribed: LINKED\_LIST [OBSERVER]}\]

To register itself, an observer may execute

\[\text{subscribe (some\_publisher)}\]

where \text{subscribe} is defined in \text{OBSERVER}:

\[
\text{subscribe (p: PUBLISHER) is}
\]

\[
\quad \text{-- Make current object observe}\ p.
\]

\[
\text{require}
\]

\[
\quad \text{publisher\_exists: } p \neq \text{Void}
\]

\[
\text{do}
\]

\[
\quad p.\text{attach (Current)}
\]

\[
\text{end}
\]
Attaching an observer

In class \textit{PUBLISHER}:\[\text{attach } (s: \textit{SUBSCRIBER}) \text{ is}\]

--- Register \textit{s} as subscriber to current publisher.

\begin{verbatim}
require
    subscriber_exists: \textit{p} /= \textit{Void}
do
    subscribed.extend \text{(s)}
end
\end{verbatim}

Note that invariant of \textit{PUBLISHER} includes the clause

\begin{verbatim}
subscribed /= \textit{Void}
\end{verbatim}

(List \textit{subscribed} is created by creation procedures of \textit{PUBLISHER})
**Triggering an event**

\[\text{trigger is} \]
\[\text{do}\]
\[\text{from subscribed.start}\]
\[\text{until subscribed.after}\]
\[\text{loop subscribed.item. update}\]
\[\text{subscribed.forth}\]
\[\text{end}\]
\[\text{end}\]

Each descendant of \textit{OBSERVER} defines its own version of \textit{update}.
Observer pattern

- Publishers know about subscribers
- Subscriber may subscribe to at most one publisher
- May subscribe at most one operation
- Not reusable — must be coded anew for each application
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.
Another approach: 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.
The EiffelVision style

```
my_button.click.action_list.extend (agent my_procedure)
```
Representing entries in the 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.
Representing entries in the action-event table

**Event:** each event *type* will be an object

*Example:* mouse clicks

**Context:** an object, usually representing element of user interface

*Example:* a particular button

**Action:** an object representing with a routine

*Example:* routine to save the file

Objects representing routines are called **agents**.
The EiffelVision style

\texttt{my\_button.click.action\_list.extend (agent my\_procedure)}
Mechanisms in other languages

- C and C++: “function pointers”

- C#: delegates (more limited form of agents)
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)
```
Subscriber variants

```plaintext
click.subscribe (agent my_procedure)

my_button. click.subscribe (agent my_procedure)

click.subscribe (agent your_procedure (a, ?, ?, b))

click.subscribe (agent other_object. other_procedure)
```
Tuples

Tuple types (for any types $A, B, C, \ldots$):

- \text{TUPLE}
- \text{TUPLE} [A]
- \text{TUPLE} [A, B]
- \text{TUPLE} [A, B, C]
- \ldots

A tuple of type \text{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. [$a_1, b_1, c_1$]
Tuple type inheritance

TUPLE

TUPLE [A]

TUPLE [A, B]
Using the Eiffel Event Library

The basic class is \textit{EVENT\_TYPE}

On the publisher side, e.g. GUI library:

- (Once) declare event type:

  \textit{click: EVENT\_TYPE [TUPLE [INTEGER, INTEGER]]}

- (Once) create event type object:

  \textit{create click}

- To trigger one occurrence of the event:

  \textit{click.publish ([x\_coordinate, y\_coordinate])}

On the subscriber side, e.g. an application:

\textit{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:

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

A tuple

If $a$ is associated with a function, $a \cdot item ([ \ldots, \ldots])$ gives the result of applying the function.
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:

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

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

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

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

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

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

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

\[ u.call(\[\]) \]

\[ v.call([a3]) \]

\[ w.call([a2]) \]

\[ x.call([a2, a3]) \]

\[ y.call([a1, a2, a3]) \]
Another example of using agents

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

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

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

\[
\text{my\_integrator\textunderscore integral\,(agent your\_function(\,), u, v), a, b)}
\]
The integration function

```
integral (f : FUNCTION [ANY, TUPLE [REAL], REAL];
        low, high: REAL): REAL is
   -- Integral of f over the interval [low, high]
   local
      x: REAL; i: INTEGER
   do
      from x := low until x > high loop
         Result := Result + f.item ([x]) * step
         i := i + 1
         x := x + i * step
      end
   end
```
Another application: using an iterator

\[ \texttt{all\_positive := my\_integer\_list}\texttt{.for\_all (agent is\_positive (?))} \]
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`
Applications of agents

- Undo-redo
- Iteration
- High-level contracts
- Numerical programming
- Introspection (finding out properties of the program itself)
End of lecture 21