Agents & event-driven programming in Eiffel

Bertrand Meyer

LASER 2012

Handling input through traditional techniques

Program drives user:

from
  i := 0
  read_line
until end_of_file loop
  i := i + 1
  Result[i] := last_line
  read_line
end
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

\[ \text{find\_station}(x, y) \]

where \( x \) and \( y \) are the mouse coordinates and \text{find\_station} is a specific procedure of your system.
Event-driven programming: a metaphor

Publishers

Subscribers

Routine

Routine

Routine

Routine

Observing a value

Subject

Observers

35.4

Temperature

Database
Alternative terminologies

- Observed / Observer
- Subject / Observer
- Publisher / Subscriber

Events and event types

Events Overview (from .NET documentation)

Events have the following properties:
1. The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
2. An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
3. Events that have no subscribers are never called.
4. Events are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces.
5. When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see [another section].
6. Events can be used to synchronize threads.
7. In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.
Event arguments

Some events are characterized just by the fact of their occurrence

Others have arguments:

- A mouse click happens at position \([x, y]\)

- A key press has a certain character code
  (if we have a single “key press” event type: we could also have a separate event type for each key)

An architectural solution: the Observer Pattern

* Deferred (abstract)
+ Effective (implemented)

Inherits from Client (uses)
Triggering an event

- Ask all observers to react to current event.

```latex
\begin{align*}
&\text{publish}^* \\
&\quad \text{-- Ask all observers to react to current event.} \\
&\text{do} \\
&\quad \text{from} \\
&\quad \quad \text{subscribed, start} \\
&\quad \text{until} \\
&\quad \quad \text{subscribed, after} \\
&\quad \text{loop} \\
&\quad \quad \text{subscribed, item, update} \\
&\quad \quad \text{subscribed, forth} \\
&\text{end} \\
&\text{end}
\end{align*}
```

Each descendant of `SUBSCRIBER` defines its own version of `update`.

---

Observer pattern (in basic form)

- Publisher objects know about subscribers
- Subscriber classes (and objects) know about their publishers
- A subscriber may subscribe to at most one publisher
- It may subscribe at most one operation
- Handling of arguments (not detailed in previous slides) requires special care
- The solution is not reusable: it must be coded anew for each application
Another approach: event-context-action table

Set of triples
[Event type, Context, Action]

Event type: any kind of event we track
   Example: left mouse click

Context: object for which these events are interesting
   Example: a particular button

Action: what we want to do when an event occurs in the context
   Example: save the file

Event-context-action table may be implemented as e.g. a hash table

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>Save_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>Map</td>
<td>Find_station</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>
Eiffel implementation using events

Simple architecture:

- One generic class: \texttt{EVENT\_TYPE}
- Two features: \textit{publish} and \textit{subscribe}

Eiffel implementation using agents

The subscribers ("observers") subscribe to events:

\begin{verbatim}
map.left\_click.subscribe(agent find\_station)
\end{verbatim}

The publisher triggers the event:

\begin{verbatim}
left\_click.publish([x\_position, y\_position])
\end{verbatim}
Observer pattern vs. Event Library

In case of an existing class $MY\_CLASS$:

- **With the Observer pattern:**
  - Need to write a descendant of $SUBSCRIBER$ and $MY\_CLASS$
  - May lead to useless multiplication of classes
  - Effect of update to call appropriate model routine

- **With the Event Library:**
  - No new classes (use library classes directly)
  - Can reuse the existing model routines directly as agents

Flexibility of the mechanism

```java
click.subscribe(agent find_station)
```

```java
map.click.subscribe(agent find_station)
```

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

```java
click.subscribe(agent other_object.other_procedure)
```
A word about tuples

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

\[
\begin{align*}
\text{TUPLE} \\
\text{TUPLE} [A] \\
\text{TUPLE} [A, B] \\
\text{TUPLE} [A, B, C] \\
\ldots
\end{align*}
\]

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. $[a1, b1, c1, d1]$
What you can do with an agent \( a \)

Call the associated routine through the feature \texttt{call} ,
whose argument is a single tuple:

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

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

Calling an agent

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

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

\[
\begin{align*}
u & := \text{agent } a0.f(a1, a2, a3) & v & := \text{agent } a0.f(a1, a2, ?) \\
v & := \text{agent } a0.f(a1, ? , a3) & x & := \text{agent } a0.f(a1, ?, ?) \\
y & := \text{agent } a0.f(?, ? , ?)
\end{align*}
\]

\[
\begin{align*}
& u.\text{call(\[])} & \text{v.\call{[a3]}} \\
& w.\call{[a2]} & x.\call{[a2, a3]} \\
& y.\call{[a1, a2, a3]}
\end{align*}
\]
Another example of using agents

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

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

\textit{my\_integrator} \cdot \text{integral} (\textit{agent my\_function}, a, b)

\textit{my\_integrator} \cdot \text{integral} (\textit{agent your\_function}(?, u, v), a, b)

The integration function

\textit{integral} (f: \text{FUNCTION}[\text{ANY, TUPLE[REAL], REAL}];
\quad \textit{a, b: REAL}: \text{REAL})

\texttt{-- Integral of f over interval [a, b].}

\texttt{local}
\quad x: \text{REAL}; i: \text{INTEGER}
\texttt{do}
\quad \text{from} \ x := a \text{ until} \ x > b \text{ loop}
\quad \text{Result} := \text{Result} + f.item([x]) \ast \text{step}
\quad i := i + 1
\quad x := a + i \ast \text{step}
\texttt{end}
\texttt{end}
Kernel library classes representing agents

Another application: using an iterator

class C feature
  all_positive, all_married: BOOLEAN

  is_positive(n: INTEGER): BOOLEAN
  -- Is n greater than zero?
  do Result := (n > 0) end

  intlist: LIST[INTEGER]
  emplist: LIST[EMPLOYEE]

end

r

do

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

  all_married := emplist. for_all(agent (EMPLOYEE) is_married)

end
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`