Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Andreas Leitner
Lecture 14: Event-driven programming and Agents
Our goal for this lecture

Extend our control structures with a more flexible mechanism, supporting in particular the needs of interactive, graphical programming (GUI)

The resulting mechanism, *agents*, has many other exciting applications

This is an Eiffel mechanism; other languages have facilities such as *delegates* (*C#*), *closures* (functional languages)
Handling input through traditional techniques

Program drives user:

\[
\begin{align*}
\text{from} & \quad i := 0 \\
\text{read_line} & \\
\text{until} & \quad \text{end_of_file loop} \\
\text{end} & \quad i := i + 1 \quad \text{Result} [i] := \text{last_line} \\
\text{end} & \quad \text{read_line}
\end{align*}
\]
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.
Some issues

1. Keeping the “business model” and the GUI separate
   - Business model (or just model): core functionality of the application
   - GUI: interaction with users

2. Minimizing “glue code” between the two

3. Making sure we keep track of what’s going on
Event-driven programming

Publishers

Subscribers

Routine

Routine

Routine

Routine
Observing a value

Observer

Subject

A = 50%
B = 30%
C = 20%
Model-View Controller
(Trygve Reenskaug, 1979)
Our 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.

CLICK START STATION ABOVE
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.
Observing a value

Observer

Subject

A = 50%
B = 30%
C = 20%
Alternative terminologies

Observed / Observer

Subject / Observer

Publish / Subscribe

Event-driven design/programming

In this presentation: Publisher and Subscriber
A solution: the Observer Pattern

* Deferred (abstract)
+ Effective (implemented)

Inherits from Client (uses)
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 is *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley 1994.
A solution: the Observer Pattern

* Deferred (abstract)
+ Effective (implemented)

Inherits from
Client (uses)
Observer pattern

Publisher keeps a (secret) list of observers:

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

To register itself, an observer executes

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

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

\[
\begin{align*}
\text{subscribe (p: PUBLISHER)} & \quad -- \text{Make current object observe } p. \\
\text{require} & \quad \text{publisher\_exists: p \neq Void} \\
\text{do} & \quad p\text{.attach (Current)} \\
\text{end}
\end{align*}
\]
Attaching an observer

In class `PUBLISHER`:

```feature {SUBSCRIBER}
    attach (s: SUBSCRIBER)
    -- Register s as subscriber to this publisher.
    require
        subscriber_exists: s /= Void
    do
        subscribed.extend(s)
    end
```

Note that the invariant of `PUBLISHER` includes the clause `subscribed /= Void` (List `subscribed` is created by creation procedures of `PUBLISHER`)
Triggering an event

publish
-- Ask all observers to
-- react to current event.
do
from subscribed.start
until subscribed.after
loop subscribed.item. update
subscribed.forth
end
end

Each descendant of OBSERVER defines its own version of update

Dynamic binding

PUBLISHER

*attach
detach

subscribed: LIST[

PUB_1

SUB_1

update*

update*

Dynamic binding

s1
s2
s3
s4

item
after

sub

Cursor
forth
Observer pattern (in basic form)

- 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
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>
Action-event table

Set of triples

\[ \text{[Event, Context, Action]} \]

**Event**: any occurrence we track
  - **Example**: a left click

**Context**: object for which the event is interesting
  - **Example**: the map widget

**Action**: what we want to do when the event occurs in context
  - **Example**: find the station closest to coordinates

Action-event table may have various implementations, e.g. hash table.
In EiffelVision

\[ \text{Paris_map} \cdot \text{click} \cdot \text{action_list} \cdot \text{extend} (\text{agent } \text{find_station}) \]
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

\[ \text{integral} (\& f, a, b) \]

where \( f \) is the function to integrate. \( \& f \) (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.
A little language quiz

What does this do?

\[
f: \text{INTEGER} \\
\text{do} \\
\quad \text{if } x > 0 \text{ then} \\
\quad \quad \text{Result} := g(f) \\
\quad \text{end} \\
\text{end} \\
\]

\[
g(x: \text{INTEGER}): \text{INTEGER} \\
\text{do} \\
\quad \text{Result} := x \\
\text{end} \\
\]
P1. Introduce new class ClickArgs inheriting from EventArgs, repeating arguments types of yourProcedure:

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

P2. Introduce new type ClickDelegate (delegate type) based on that class

```csharp
public void delegate ClickDelegate (Object sender, ClickArgs e);
```

P3. Declare new type Click (event type) based on the type ClickDelegate:

```csharp
public event ClickDelegate Click;
```
P4. Write new procedure `OnClick` to wrap handling:

```csharp
protected void OnClick (Clickargs c)
{
    if (Click != null) {Click (this, c);}
}
```

P5. For every event occurrence, create new object (instance of `ClickArgs`), passing arguments to constructor:

```csharp
ClickArgs yourClickargs = new Clickargs (h, v);
```

P6. For every event occurrence, trigger event:

```csharp
OnClick (yourClickargs);
```
With .NET delegates: subscriber

D1. Declare a delegate `myDelegate` of type `ClickDelegate`. (Usually combined with following step.)

D2. Instantiate it with `yourProcedure` as argument:

   ```
   myDelegate = new ClickDelegate (yourProcedure);
   ```

D3. Add it to the delegate list for the event:

   ```
   YES_button.Click += myDelegate;
   ```
Using the Eiffel approach (Event Library)

- **Event**: each event type will be an object
  - **Example**: left click

- **Context**: an object, usually representing a user interface element
  - **Example**: the map

- **Action**: an agent representing a routine
  - **Example**: `find_station`
The Event library

Basically:

- One generic class: EVENT_TYPE
- Two features: publish and subscribe

For example: A map widget Paris_map that reacts in a way defined in find_station when clicked (event left_click):
Example using the Event library

The publisher ("subject") creates an event type object:

```plaintext
left_click: EVENT_TYPE [ TUPLE [INTEGER, INTEGER]]
   -- Left mouse click events
once
   create Result
ensure
   exists: Result /= Void
end
```

The publisher triggers the event:

```plaintext
left_click.publish ([x_position, y_position])
```

The subscribers ("observers") subscribe to events:

```plaintext
Paris_map.left_click.subscribe (agent find_station)
```
Event Library style

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 find_station)`
Observer pattern vs. Event Library

In case of an existing class $\textit{MY\_CLASS}$:

- With the Observer pattern:
  - Need to write a descendant of $\textit{SUBSCRIBER}$ and $\textit{MY\_CLASS}$
  - $\Rightarrow$ Useless multiplication of classes

- With the Event Library:
  - Can reuse the existing routines directly as agents
Subscriber variants

\[
\text{click.subscribe (agent find_station)}
\]

\[
\text{Paris_map. click.subscribe (agent find_station)}
\]

\[
\text{click.subscribe (agent your_procedure (a, ?, ?, b))}
\]

\[
\text{click.subscribe (agent other_object.other_procedure)}
\]
Tuples

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

- `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, d1]`
Labeled tuple types

**TUPLE [author: STRING; year: INTEGER; title: STRING]**

Restricted form of class

A labeled tuple type denotes the same type as an unlabeled form, here

**TUPLE [STRING, INTEGER, STRING]**

but facilitates access to individual elements

To denote a particular tuple (labeled or not):

["Dostoievsky", 1865, "War and Peace"]

To access tuple elements: use e.g. \texttt{t.year}
What you can do with an agent \( a \)

Call the associated routine through the feature “\( call \)”, whose argument is a single tuple:

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

A tuple

If \( a \) is associated with a function, \( a \cdot \text{item} ([\ldots, \ldots]) \) gives the result of applying the function.
Features applicable to an agent $a$:

- If $a$ represents a procedure, $a.call([argument\_tuple])$ calls the procedure.

- If $a$ represents a function, $a.item([argument\_tuple])$ calls the function and returns its result.
Tuple type inheritance

TUPLE

TUPLE[A]

TUPLE[A, B]
Using the Eiffel Event Library

The basic class is `TRAFFIC_EVENT_CHANNEL`

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

- (Once) declare event type:

  ```
  click: TRAFFIC_EVENT_CHANNEL
  [ 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 find_station)
```
What you can do with an agent $a$

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

$$a\text{.call}([\text{horizontal}\_\text{position}, \text{vertical}\_\text{position}])$$

If $a$ is associated with a function, $a\text{.item}([\ldots, \ldots])$ 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:

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

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

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

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

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

\[ f(x_1: T_1; x_2: T_2; x_3: T_3) \]
\[ a_0: C, a_1: T_1; a_2: T_2; a_3: T_3 \]

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

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

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

\[ x := \text{agent } a_0. f(a_1, ?, ?) \]
\[ x.call([a_2, a_3]) \]

\[ y := \text{agent } a_0. f(?, ?, ?) \]
\[ y.call([a_1, a_2, a_3]) \]
Calling the agent

\[ f(x_1: T_1; x_2: T_2; x_3: T_3) \]
\[ a_0: C; \; a_1: T_1; \; a_2: T_2; \; a_3: T_3 \]

\[ \begin{align*}
    u & := \text{agent } a_0. f(a_1, a_2, a_3) & & u.\text{call}([],) \\
    v & := \text{agent } a_0. f(a_1, a_2, ?) & & v.\text{call}([a_3]) \\
    w & := \text{agent } a_0. f(a_1, ?, a_3) & & w.\text{call}([a_2]) \\
    x & := \text{agent } a_0. f(a_1, ?, ?) & & x.\text{call}([a_2, a_3]) \\
    y & := \text{agent } a_0. f(?, ?, ?) & & y.\text{call}([a_1, a_2, a_3])
\end{align*} \]
Another example of using agents

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

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

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

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

\[
\text{integral} \left( f : \text{FUNCTION}[\text{ANY, TUPLE[REAL], REAL}]; \right.
\]
\[
\text{low, high: REAL}: \text{REAL}
\]
\[
\quad \text{-- Integral of } f
\]
\[
\quad \text{-- over the interval [low, high]}
\]
\[
\text{local}
\]
\[
x: \text{REAL}; i: \text{INTEGER}
\]
\[
do
\]
\[
\text{from } x := \text{low until } x > \text{high loop}
\]
\[
\text{Result} := \text{Result} + f.\text{item}([x]) \ast \text{step}
\]
\[
i := i + 1
\]
\[
x := a + i \ast \text{step}
\]
\[
\text{end}
\]
\[
\text{end}
\]
Another application: using an iterator

\[ all\_positive := \text{my}\_\text{integer}\_\text{list}\.\text{for}\_\text{all} (\text{agent is}\_\text{positive} (?)) \]
Iterators

In class \textit{LINEAR} \([G]\), ancestor to all classes for lists, sequences etc., you will find:

\begin{verbatim}
for_all
there_exists
do_all
do_if
do_while
do_until
\end{verbatim}
Applications of agents

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

call

ROUTINE

* deferred
+ effective

PROCEDURE

FUNCTION

+ item
Declaring an agent

\[ p: \text{PROCEDURE} [\text{ANY}, \text{TUPLE}] \]
\[ \quad \text{-- Agent representing a procedure,} \]
\[ \quad \text{-- no open arguments} \]

\[ q: \text{PROCEDURE} [\text{ANY}, \text{TUPLE} [X, Y, Z]] \]
\[ \quad \text{-- Agent representing a procedure,} \]
\[ \quad \text{-- 3 open arguments} \]

\[ f: \text{FUNCTION} [\text{ANY}, \text{TUPLE} [X, Y, Z], \text{RES}] \]
\[ \quad \text{-- Agent representing a procedure,} \]
\[ \quad \text{-- 3 open arguments, result of type } \text{RES} \]
Calling the agent

\[ f(x_1: T_1; x_2: T_2; x_3: T_3) \]
\[ a_0: C; a_1: T_1; a_2: T_2; a_3: T_3 \]

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

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

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

\[
x := \text{agent } a_0. f(a_1, ?, ?) \]
\[ x.\text{call}([a_2, a_3]) \]

\[
y := \text{agent } a_0. f(? , ? , ?) \]
\[ y.\text{call}([a_1, a_2, a_3]) \]