Event-Driven programming, agents & tuples
Our goal for this session

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

Other languages have facilities such as delegates (C#), closures (functional languages)
Handling input through traditional techniques

Program drives user:

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

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

where \( x \) and \( y \) are the mouse coordinates and \textit{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: a metaphor

Publishers

Subscribers

Routine

Routine

Routine

Routine
Observing a value

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 \textit{find\_station} is a specific procedure of your system.
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.
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)
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 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}:

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

\[
\text{do}
\]

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

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

In class \textit{PUBLISHER}:

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

Note that the invariant of \textit{PUBLISHER} includes the clause
\begin{verbatim}
subscribed /= Void
\end{verbatim}

(List \textit{subscribed} is created by creation procedures of \textit{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
```
Observer pattern (in basic form)

- Publishers know about subscribers

- May subscribe at most one operation (corresponding to one event type)

- Handling arguments is messy

- 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-context-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>
Event-action-context table

Set of triples

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

**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

Paris_map.click.action_list.extend(agent find_station)
Mechanisms in other languages

C and C++: “function pointers”

C#: delegates (more limited form of agents)
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?

\[ f : \text{INTEGER} \]
\[
\text{do}
\]
\[
\text{Result} := g(f)
\]
\[
\text{end}
\]

\[ g(x : \text{INTEGER}) : \text{INTEGER} \]
\[
\text{do}
\]
\[
\text{Result} := x
\]
\[
\text{end}
\]
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 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

Basis:

- 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:

\[\text{left\_click} : \text{EVENT\_TYPE} [\text{TUPLE} [\text{INTEGER}, \text{INTEGER}]]\]

\[\text{-- Left mouse click events}\]

\[
\begin{align*}
\text{once} \\
\text{create Result} \\
\text{ensure} \\
\text{exists: Result} /= \text{Void} \\
\text{end}
\end{align*}
\]

The publisher triggers the event:

\[\text{left\_click}.\text{publish} ([\text{x\_position}, \text{y\_position}])\]

The subscribers ("observers") subscribe to events:

\[\text{Paris\_map\_left\_click}.\text{subscribe (agent find\_station)}\]
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}
  - Useless multiplication of classes

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

\texttt{click.subscribe(agent find\_station)}

\texttt{Paris\_map. click.subscribe(agent find\_station)}

\texttt{click.subscribe(agent your\_procedure(a, ?, ?, b))}

\texttt{click.subscribe(agent other\_object.\_other\_procedure)}
Tuples

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

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

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.

$[a1, b1, c1, d1]$
Tuple type inheritance

- TUPLE
- TUPLE [A]
- TUPLE [A, B]
- ...

Diagram showing the inheritance of tuple types.
Labeled tuple types

\[ TUPLE \{ \text{author} : \text{STRING} ; \text{year} : \text{INTEGER} ; \text{title} : \text{STRING} \} \]

Restricted form of class

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

\[ TUPLE \{ \text{STRING} , \text{INTEGER} , \text{STRING} \} \]

but facilitates access to individual elements

To denote a particular tuple (labeled or not):

\[ ["\text{Tolstoi}" , 1865 , "\text{War and Peace}" ] \]

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

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

\begin{verbatim}
  a.call([horizontal_position, vertical_position])
\end{verbatim}

If \textit{a} is associated with a function, \textit{a.item([ ..., ...])} gives the result of applying the function.
Tuples: Procedures vs. Functions

Features applicable to an agent $a$:

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

- If $a$ represents a function, $a.\text{item}([\text{argument\_tuple}])$ calls the function and returns its result
Example using the Event library

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

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

The publisher triggers the event:

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

The subscribers ("observers") subscribe to events:

```
Paris_map.left_click.subscribe (agent find_station)
```
What you can do with an agent $a$

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

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

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

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

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

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

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

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

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

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

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

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

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

\[
y := \text{agent } a_0. f(?, ?, ?) \]
\[ y.\text{call }([a_1, a_2, a_3]) \]
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} \cdot \text{integral(\textbf{agent my\_function}, a, b)}

\text{my\_integrator} \cdot \text{integral(\textbf{agent your\_function}(?, u, v), a, b)}
The integration function

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

-- Integral of \( f \) over interval \([a, b]\)

local
\[ x : \text{REAL}; i : \text{INTEGER} \]

do

from \( x := a \) until \( x > b \) loop
\[ \text{Result := Result + } f.\text{item ([x]) \ast step} \]
\[ i := i + 1 \]
\[ x := a + i \ast \text{step} \]
end

end
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]

r
    do
        all_positive := intlist. for_all (agent is_positive (?))
        all_married := emplist. for_all (agent {EMPLOYEE} is_married)
    end
end
Reminder: using inline agents

```plaintext
intlist.for_all
  (agent (x: INTEGER): BOOLEAN
    do
      Result := (x > 0)
    end)
```
Iterators

In class $\text{LINEAR}[G]$, 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

Patterns: Observer, Visitor, Undo-redo (command)
Iteration
High-level contracts
Numerical programming
Introspection (finding out properties of the program itself)
Kernel library classes representing agents

call ROUTINE

+ PROCEDURE

+ FUNCTION

last_result

item

+ PREDICATE
Declaring an agent

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

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

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

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

\[ u := \text{agent } a_0 \cdot f(a_1, a_2, a_3) \]
\[ v := \text{agent } a_0 \cdot f(a_1, a_2, ?) \]
\[ w := \text{agent } a_0 \cdot f(a_1, ?, a_3) \]
\[ x := \text{agent } a_0 \cdot f(a_1, ?, ?) \]
\[ y := \text{agent } a_0 \cdot f(?, ?, ?) \]

\[ u.\text{call}([]) \]
\[ v.\text{call}([a_3]) \]
\[ w.\text{call}([a_2]) \]
\[ x.\text{call}([a_2, a_3]) \]
\[ y.\text{call}([a_1, a_2, a_3]) \]
Agents: summary and conclusion

Treat routines (computation) as objects

An indispensable complement to basic O-O mechanisms