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:

```plaintext
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.

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

Publishers: Routine
Subscribers: Routine

Observing a value

Observer: A = 50%, B = 30%, C = 20%

Subject: A = 50%, B = 30%, C = 20%

Model-View Controller

(Trygve Reenskaug, 1979)
Intro. to Programming, lecture 23: Event-driven programming and Agents

Our example

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

\texttt{find\_station}(x, y)

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

Confusion

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 \texttt{EventHandler} delegate and the \texttt{EventArgs} base class.

Observing a value

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)

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

![Diagram of Observer Pattern]

**Deferred (abstract)**

**Effective (implemented)**

Inherits from

Client (uses)

---

Observer pattern

Publisher keeps a (secret) list of observers:

`subscribed: LINKED_LIST(SUBSCRIBER)`

To register itself, an observer executes

`subscribe(some_publisher)`

where `subscribe` is defined in `SUBSCRIBER`:

```ruby
subscribe(p: PUBLISHER)  
-- Make current object observe p.
  require
  do publisher_exists: p /= Void
  p.attach(Current)
  end
```

Attaching an observer

In class `PUBLISHER`:

```ruby
feature (SUBSCRIBER)
  attach(s: SUBSCRIBER)  
  -- Register s as subscriber to this publisher.
    require
    do 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`)
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

<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

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

- **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)

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?

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

\[
g(x: \text{INTEGER}) : \text{INTEGER}
do\begin{align*}
\text{Result } := x \\
\text{end}
\end{align*}
\]
**With .NET delegates: publisher (1)**

<table>
<thead>
<tr>
<th>P1</th>
<th>Introduce new class ClickArgs inheriting from EventArgs, repeating arguments types of yourProcedure:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public class ClickArgs { int x, y; }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P2</th>
<th>Introduce new type ClickDelegate (delegate type) based on that class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public void delegate ClickDelegate (Object sender, ClickArgs e);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P3</th>
<th>Declare new type Click (event type) based on the type ClickDelegate:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public event ClickDelegate Click;</td>
</tr>
</tbody>
</table>

**With .NET delegates: publisher (2)**

<table>
<thead>
<tr>
<th>P4</th>
<th>Write new procedure OnClick to wrap handling:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>protected void OnClick (ClickArgs c) {</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P5</th>
<th>For every event occurrence, create new object (instance of ClickArgs), passing arguments to constructor:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ClickArgs yourClickargs = new ClickArgs (h, v);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P6</th>
<th>For every event occurrence, trigger event:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OnClick (yourClickargs);</td>
</tr>
</tbody>
</table>

**With .NET delegates: subscriber**

<table>
<thead>
<tr>
<th>D1</th>
<th>Declare a delegate myDelegate of type ClickDelegate. (Usually combined with following step.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>myDelegate = new ClickDelegate (yourProcedure);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D2</th>
<th>Instantiate it with yourProcedure as argument:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>myDelegate += new ClickDelegate (yourProcedure);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D3</th>
<th>Add it to the delegate list for the event:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>YES_button.Click += myDelegate;</td>
</tr>
</tbody>
</table>
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:

```eiffel
define left_click: EVENT_TYPE
once
ensure
exists: Result /= Void
```

The publisher triggers the event:

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

The subscribers ("observers") subscribe to events:

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

The basic class is \texttt{EVENT\_TYPE}

On the publisher side, e.g. GUI library:
- (Once) declare event type:
  \texttt{click \texttt{\textbackslash{}texttt{\texttt{\textbackslash{}texttt{EVENT\_TYPE}}}}} \texttt{\texttt{(\texttt{TUPLE}} \texttt{\texttt{\texttt{\texttt{INTEGER, INTEGER}})}}
- (Once) create event type object:
  \texttt{create click}
- To trigger one occurrence of the event:
  \texttt{click.publish([x\_coordinate, y\_coordinate])}

On the subscriber side, e.g. an application:
\texttt{click.subscribe(agent find\_station)}

Observer pattern vs. Event Library

In case of an existing class \texttt{MY\_CLASS}:
- With the Observer pattern:
  - Need to write a descendant of \texttt{SUBSCRIBER} and \texttt{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 \texttt{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 \(A, B, C, \ldots\)):  

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

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. \([a_1, b_1, c_1, d_1]\)

Labeled tuple types

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

Restricted form of class

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

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

but facilitates access to individual elements

To denote a particular tuple (labeled or not):

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

To access tuple elements: use e.g. \(t.\text{year}\)

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, \(a.\text{item}([\ldots])\) gives the result of 
applying the function.
**Tuples: Procedures vs. Functions**

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 type inheritance diagram]

**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 \texttt{a}

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

\begin{align*}
\texttt{a.call([[\text{horizontal\_position}, \text{vertical\_position}]])}
\end{align*}

A tuple

If \texttt{a} is associated with a function, \texttt{a.item([..., ...])} 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*}
w & := \texttt{agent a0.f(a1, a2, a3)} \quad \text{-- All closed (as before)} \\
w & := \texttt{agent a0.f(a1, a2, ?)} \\
x & := \texttt{agent a0.f(a\ldots, a3)} \\
y & := \texttt{agent a0.f(a\ldots, a\ldots)} \\
z & := \texttt{agent a0.f(?\ldots, ?)}
\end{align*}

Calling the agent

\begin{align*}
f(x1: T1; x2: T2; x3: T3) \\
an: C; a1: T1; a2: T2; a3: T3
\end{align*}

\begin{align*}
u & := \texttt{agent a0.f(a1, a2, a3)} \\
u & := \texttt{x.call([])} \\
v & := \texttt{agent a0.f(a1, a2, ?)} \\
v & := \texttt{x.call([a2])} \\
w & := \texttt{agent a0.f(a1?, a2)} \\
x & := \texttt{x.call([a2])} \\
y & := \texttt{agent a0.f(?\ldots, ?)} \\
y & := \texttt{x.call([x1, a2, a3])}
\end{align*}
Calling the agent

\[
f(x_1: T_1; x_2: T_2; x_3: T_3)
\]
\[
a_0: C; \quad a_1: T_1; \quad a_2: T_2; \quad a_3: T_3
\]
\[
u := \text{agent } a_0, f(a_1, a_2, a_3) \quad \text{u.call([])}
\]
\[
v := \text{agent } a_0, f(a_1, a_2, ?) \quad \text{v.call([a_2])}
\]
\[
w := \text{agent } a_0, f(a_1, ?, a_3) \quad \text{w.call([a_2])}
\]
\[
x := \text{agent } a_0, f(? , ?, a_3) \quad \text{x.call([a_2, a_3])}
\]
\[
y := \text{agent } a_0, f(? , ?, ?) \quad \text{y.call([a_1, a_2, a_3])}
\]

Another example of using agents

\[
\int_{a}^{b} \text{my_function} (x) \, dx
\]
\[
\int_{a}^{b} \text{your_function} (x, u, v) \, dx
\]
\[
\text{my_integrator.integral (agent my_function , a, b)}
\]
\[
\text{my_integrator.integral (agent your_function (x, u, v), a, b)}
\]

The integration function

\[
\text{integral (} f, \text{FUNCTION}[\text{ANY, TUPLE[REAL, REAL]]};
\]
\[
\text{low, high: REAL, REAL} \quad \text{REAL}
\]
\[
\text{local x: REAL, i: INTEGER}
\]
\[
do \text{from } x := \text{low} \text{ until } x > \text{high} \text{ loop}
\]
\[
\text{Result := Result + } f(x, \text{item ([x])}) \times \text{step}
\]
\[
i := i + 1
\]
\[
x := a + i \times \text{step}
\]
\[
end
\]
\[
end
\]
Another application: using an iterator

\[
\text{all_positive := my_integer_list.
for_all (agent is_positive(?))}
\]

Iterators

In class \textsc{Linear[6]}, ancestor to all classes for lists, sequences etc., you will find:

- \textit{for_all}
- \textit{there_exists}
- \textit{do_all}
- \textit{do_if}
- \textit{do_while}
- \textit{do_until}

Applications of agents

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

Declarating an agent

\begin{align*}
\text{p: PROCEDURE[ANY, TUPLE]} & \quad -- \text{Agent representing a procedure,} \\
& \quad -- \text{no open arguments} \\
\text{q: PROCEDURE[ANY, TUPLE[X, Y, Z]]} & \quad -- \text{Agent representing a procedure,} \\
& \quad -- \text{3 open arguments} \\
\text{f: FUNCTION[ANY, TUPLE[X, Y, Z], RES]} & \quad -- \text{Agent representing a procedure,} \\
& \quad -- \text{3 open arguments, result of type RES}
\end{align*}

Calling the agent

\begin{align*}
f(x1; x2; x3; x4) & \quad a0; a1; a2; a3; a4; a5 \\
& \text{u := agent a0, f(a1, a2, a3)} \\
& \text{v := agent a0, f(a1, a2, ?)} \\
& \text{w := agent a0, f(a1, ?, a3)} \\
& \text{x := agent a0, f(a1, ?, ?)} \\
& \text{y := agent a0, f(?, ?, ?)}
\end{align*}