Handling traditional input

Program drives input:

```java
from
read_next_character
until last_character = Enter loop
i := i + 1
Result.put (last_character, i)
read_next_character
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:

\[ \text{your\_procedure}(x, y) \]

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

---

**Avoiding glue code**

Event producer (e.g. GUI)

Direct subscription

Connection objects

Business model (application logic)

Model View Controller (MVC) Design Pattern

---

**Event-driven programming: an example**

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

\[ \text{your\_procedure}(x, y) \]

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

Publishers

Subscribers

A solution: Observer Pattern

PUBLISHER

attach
detach

GUI_CLASS

APP_CLASS

OBSERVER

update*

update+

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 are by Gamma et al. and by Feree.
A solution: Observer Pattern

Publisher keeps a list of observers:

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

To register itself, an observer may execute

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

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

\[ \text{subscribe (p: PUBLISHER)} \rightarrow \text{Make current object observe } p. \]

\[ \begin{align*}
\text{require} & \quad \text{publisher\_exists}\ p \neq \text{Void} \\
\text{do} & \quad p.\text{attach}(\text{Current}) \\
\text{end}
\end{align*} \]

Attaching an observer

In class \text{PUBLISHER}:

\[ \text{attach (s: OBSERVER)} \rightarrow \text{Register } s \text{ as subscriber to current publisher.} \]

\[ \begin{align*}
\text{require} & \quad \text{subscriber\_exists}\ s \neq \text{Void} \\
\text{do} & \quad \text{subscribe} . \text{extend}(s) \\
\text{end}
\end{align*} \]

The invariant of \text{PUBLISHER} includes the clause

\[ \text{subscribed} \neq \text{Void} \]

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

```
trigger is  -- Ask all observers to
    -- react to current event.
do
    from subscribed.start
    until subscribed.after
    loop
        subscribed.item, update
    end
end
```

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

**Observer pattern: some limitations**

- Each publisher object knows about its observers.
- Only one update procedure in `OBSERVER`.
  - Subscribe to at most one publisher.
  - At most one operation.
- Not reusable — must be coded anew for each application.
- (This is the difference between patterns and components)

**Another approach: 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>Yes_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>...</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

[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.

---

**In EiffelVision**

```
OK_button.click, action_list.extend
(agent your_procedure)
```

---

**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.

With .NET delegates: publisher (1)

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, e);
```

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

```csharp
public event ClickDelegate Click;
```

With .NET delegates: publisher (2)

P4. Write new procedure OnClick to wrap handling:

```csharp
protected void OnClick (ClickArgs e)
```

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:

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

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

```csharp
YES_button.Click += myDelegate;
```

Using the Eiffel approach (Event Library)

- **Event:** each event type will be an object
  - **Example:** mouse clicks

- **Context:** an object, usually representing a user interface element
  - **Example:** a particular button

- **Action:** an agent representing a routine
  - **Example:** `your_function`

Action-event table

Set of triples

- **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.
Using the Event Library

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

```
click.subscribe (agent your_procedure)
```  

```
my_button.click.subscribe (agent your_procedure)
```  

```
click.subscribe (agent my_procedure (a, ?, ?, b, ))
```  

```
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]
3. Accessing tuple elements

If \( t \) is a tuple \([a, b, c, \ldots]\), use

- \( t\.item(i) \) to obtain \( i \)-th element
  - May need assignment attempt:
    
    \[
    x := t\.item(i)
    \]

- \( t\.put(x, i) \) to change \( i \)-th element

4. Tuple type inheritance

5. Using the Eiffel Event Library

The basic class is \( \text{EVENT\_TYPE} \)

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

- (Once) declare event type:
  
  \[
  \text{click \: EVENT\_TYPE\{}TUPLE\{INTEGER, INTEGER\}\}}\]

- (Once) create event type object:
  
  \[
  \text{create click}\n  \]

- To trigger one occurrence of the event:
  
  \[
  \text{click\_publish\{}(x\_coordinate, y\_coordinate)\}\}}\]

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

\[
\text{click\_subscribe\{}(agent my\_procedure)\}\}}\]
What you can do with an agent \( a \)

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

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

A tuple

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

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

Calling the agent

\[
\begin{align*}
\text{f}(x1, T1, x2, T2, x3, T3) \\
a0, C, a1, T1, a2, T2, a3, T3
\end{align*}
\]

\[
\begin{align*}
u & := \text{agent } a0, f(a1, a2, a3) \\
v & := \text{agent } a0, f(a1, a2, ?) \\
w & := \text{agent } a0, f(?, a2, a3) \\
x & := \text{agent } a0, f(a1, ?, ?) \\
y & := \text{agent } a0, f(?, ?, ?)
\end{align*}
\]
Another example of using agents

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

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

\[
my\_integrator\_integral (agent my\_function, a, b)
\]

\[
my\_integrator\_integral (agent your\_function (x, u, v), a, b)
\]

The integration function

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

\[
\text{local}
\]

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

\[
\text{do}
\]

\[
\text{from } x := \text{low} \text{ until } x > \text{high} \text{ loop}
\]

\[
\text{Result} := \text{Result} + f.\text{item}((x)) \times \text{step}
\]

\[
i := i + 1
\]

\[
x := a + i \times \text{step}
\]

\[
\text{end}
\]

Another application: using an iterator

\[
\text{all_positive} := my\_integer\_list \text{ for_all} (agent is_positive (?))
\]
Iterators

In class \texttt{LINEAR[ \mathcal{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

- \texttt{call} \texttt{ROUTINE}
- \texttt{PROCEDURE}
- \texttt{FUNCTION}
Declaring an agent

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

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

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

Calling the agent

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

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

End of lecture 20