Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 15: Programmazione con gli eventi

Handling input through traditional techniques

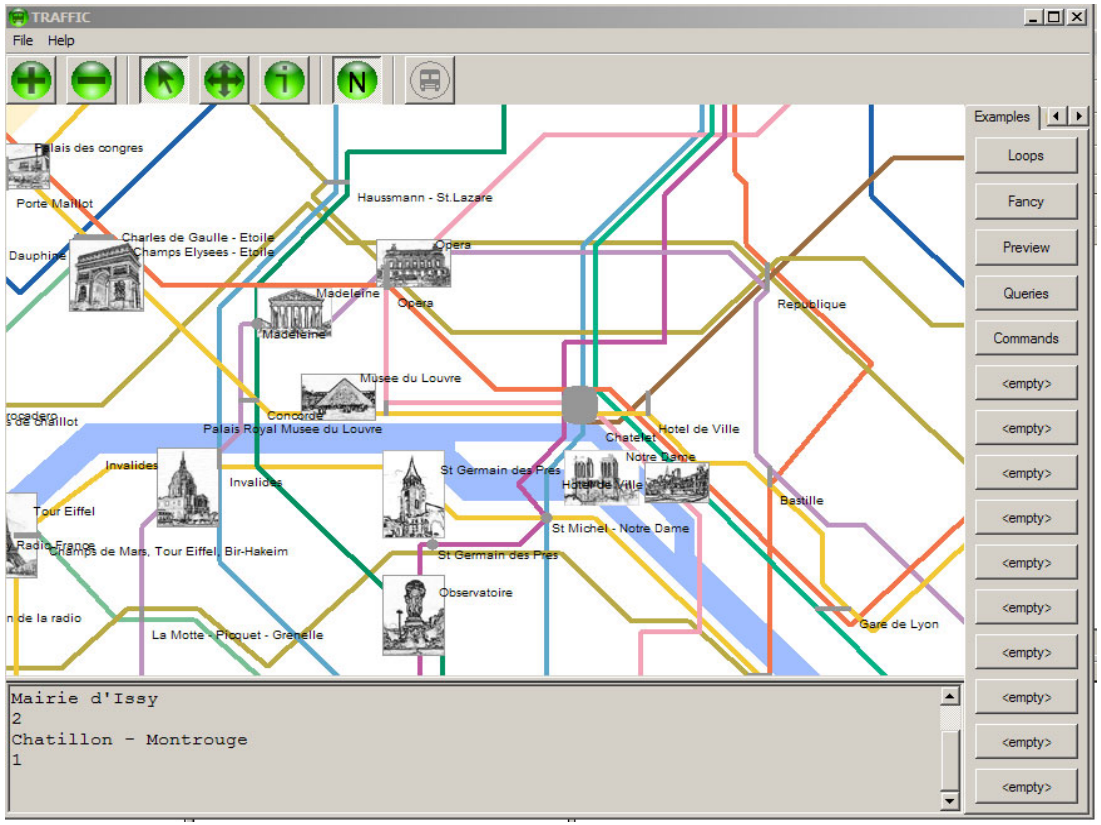
Program drives user:

```
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"



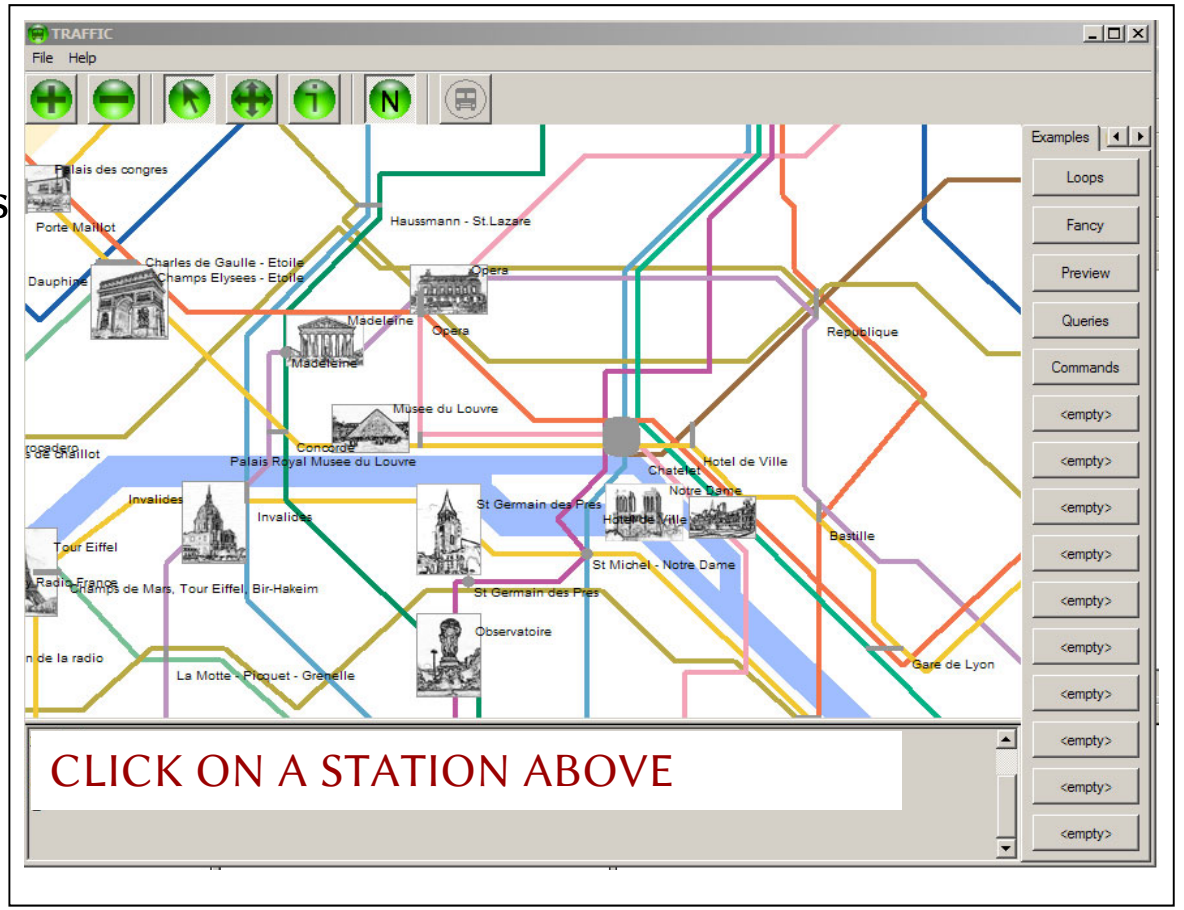
Approach: event-driven programming

An example:

Specify that when a user clicks on a station the system must execute

find_station (*x*, *y*)

where *x* and *y* are the mouse coordinates and *find_station* is a specific procedure of your system.



Event characteristics

Any event releases some information

In any case their occurrence is an information

In some cases it is the only information

In any case the event does **not** know

who will use the information, and

when it will be used

otherwise it would be a feature call

Event arguments

Some events are characterized just by the fact of their occurrence

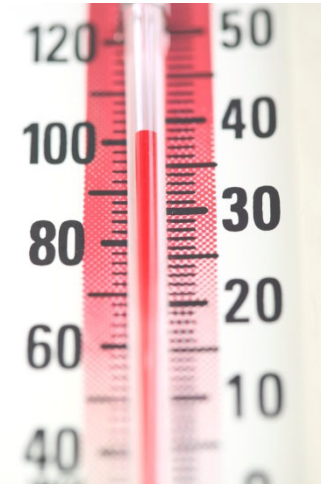
Other events have **arguments**:

- A mouse click happens at position $[x, y]$
- A key press has a certain character code
  (when we have a single "key press" event type:
  but we could also have a separate event type
  for each key)

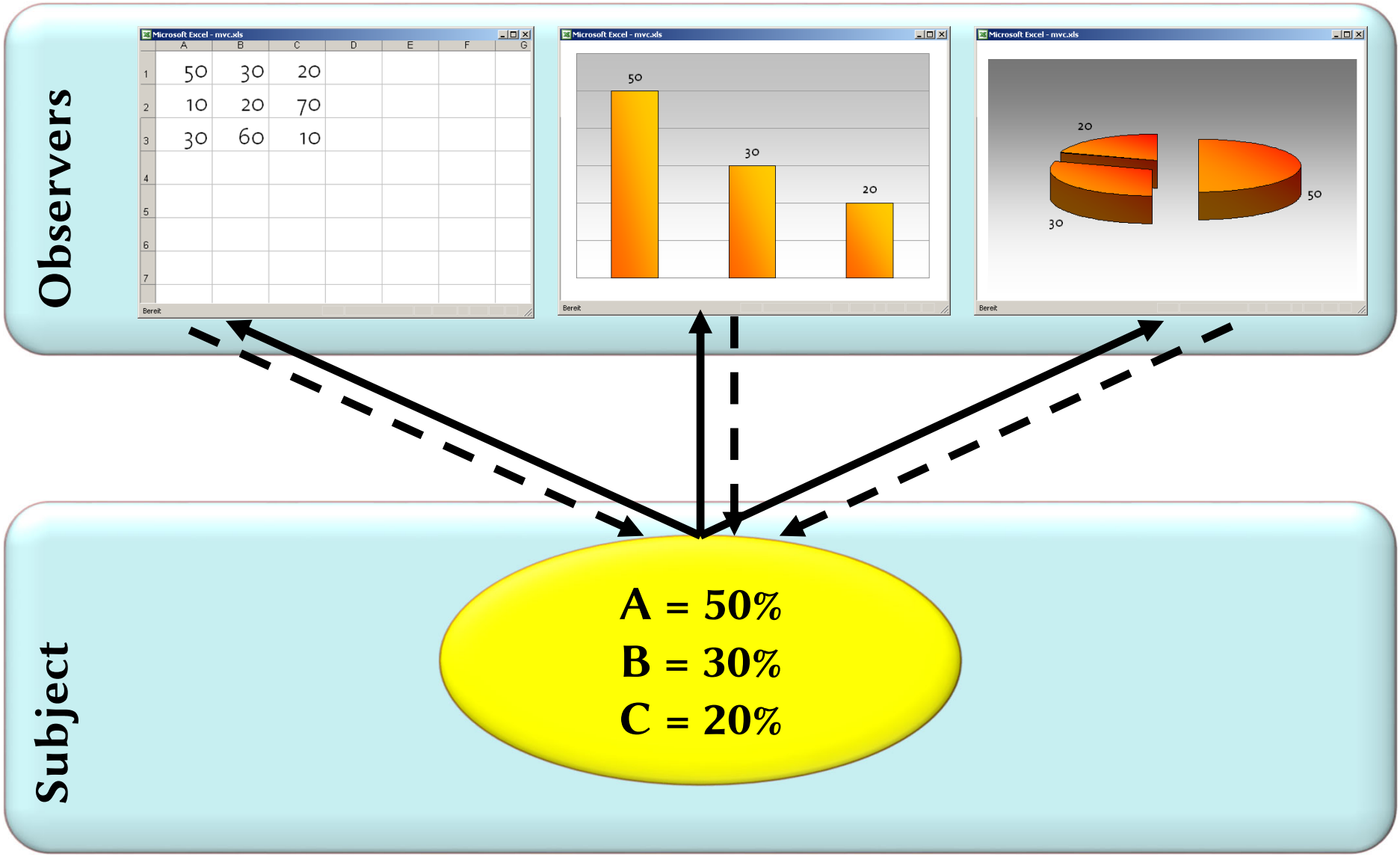
Observing a value

Subject

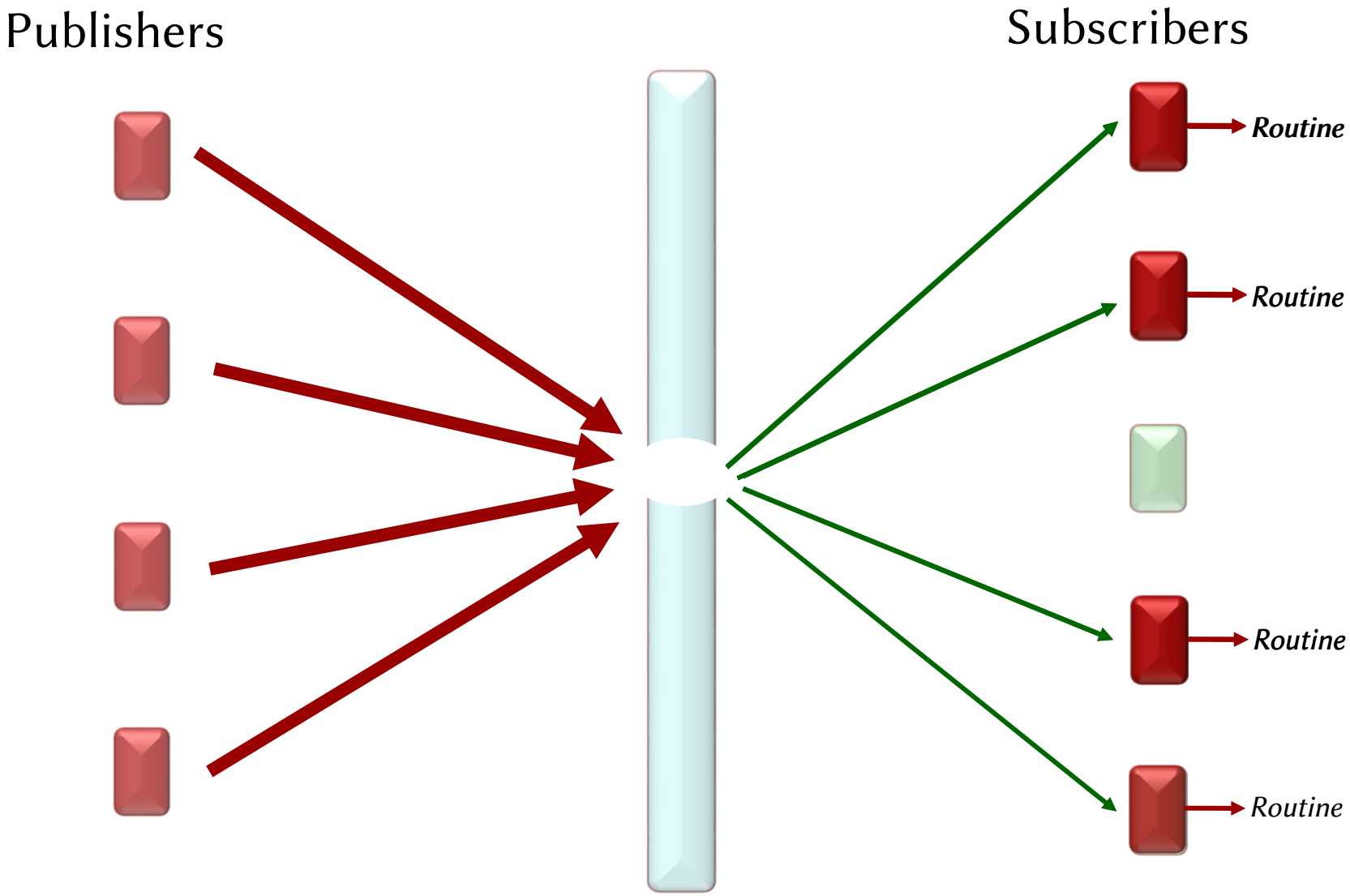
Observers



Observing a value



The general scheme



How to connect events and reactions?

Publishers will release events

Subscribers will react to events

How subscribers can learn about events?

How publishers can alert about events?

1. Polling on publishers by subscribers

2. Notification to subscribers by publishers

The Observer design pattern is an example of 2.

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

Alternative terminologies

Observed / Observer

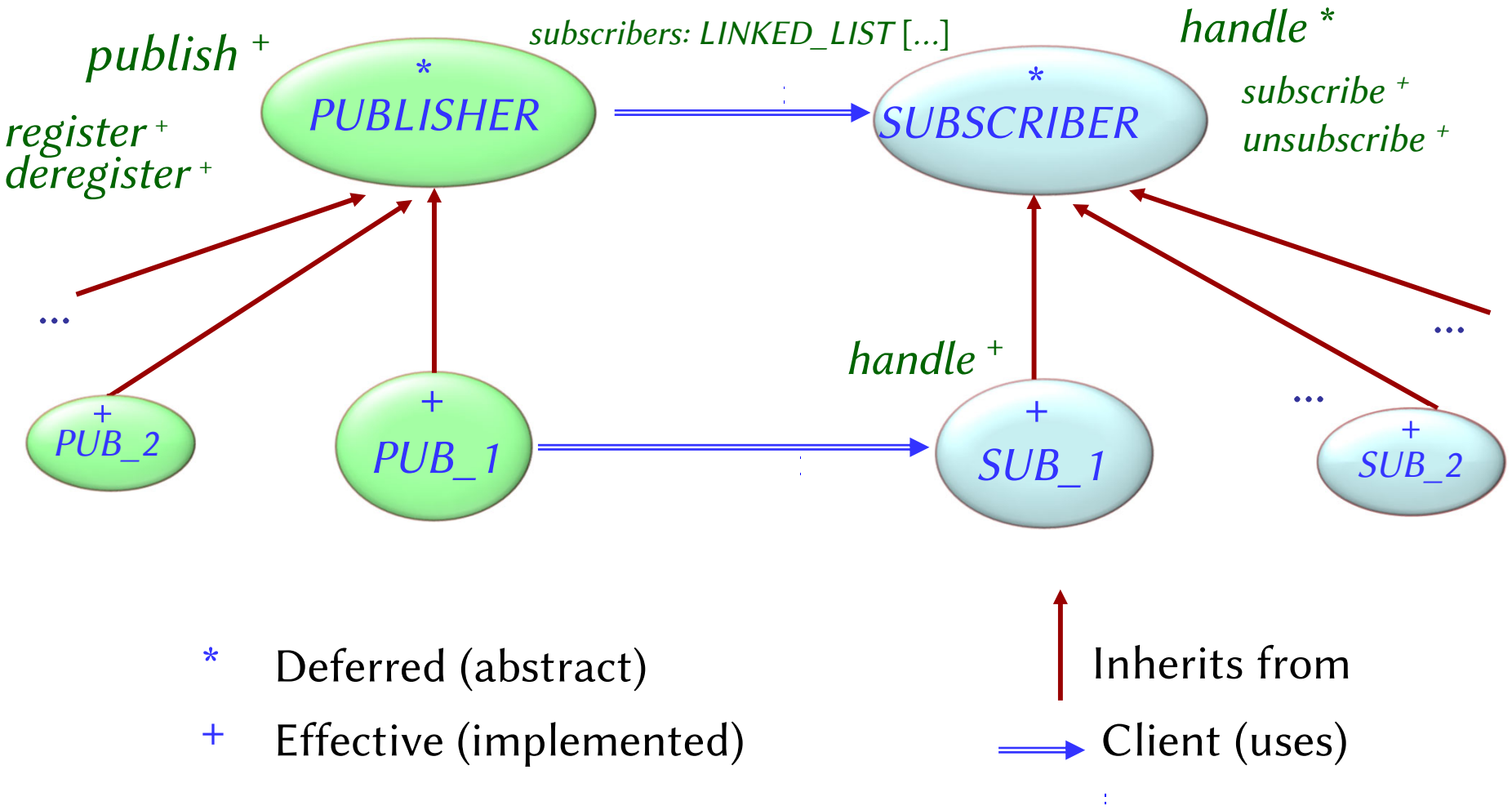
Subject / Observer

Publisher / Subscriber

Observable / Observer (in Java)

In this presentation we use Publisher and Subscriber for the concepts, Observer for the design pattern

An architectural solution: the Observer Pattern

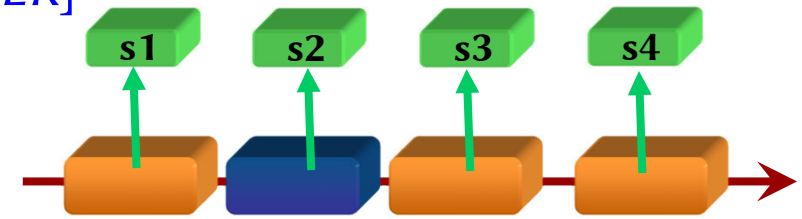


PUBLISHER class

Publisher keeps a (secret) list of subscribers:

subscribers : *LINKED_LIST* [*SUBSCRIBER*]

Why?




```
feature {SUBSCRIBER}
    register (s: SUBSCRIBER)
            -- Register s as subscriber to this publisher.
        do
            subscribers.extend (s)
        end
```

Multiple subscriptions?!

Performance problems

(Void safety will require *subscribers* is created by creation procedures of *PUBLISHER*)

deregister is implemented as list removal

SUBSCRIBER class

To register itself, a subscriber executes

```
feature subscribe (p: PUBLISHER)
            -- Make current object observe p.

        do
            p.register (Current)
        end
```

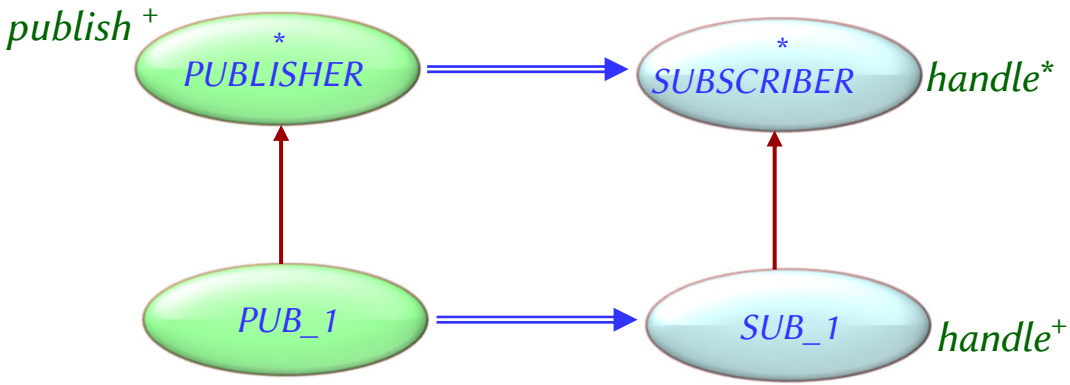
To unsubscribe itself, a subscriber executes a similar procedure

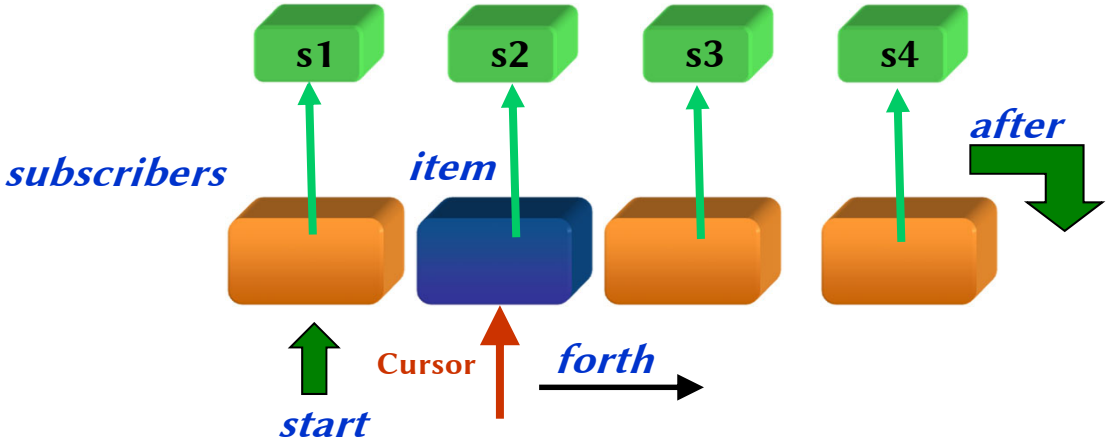
Handling events will have to be managed in subclasses:


```
feature {PUBLISHER}
    handle (args: LIST [ANY])
                -- React to publication of one event.
            deferred
        end
```

Publishing an event (1)

```
feature publish
        -- Ask all subscribers to
        -- react to current event.
    do
        from
            subscribers.start
        until
            subscribers.after
        loop
            subscribers.item.handle(args)
            subscribers.forth
        end
    end
```

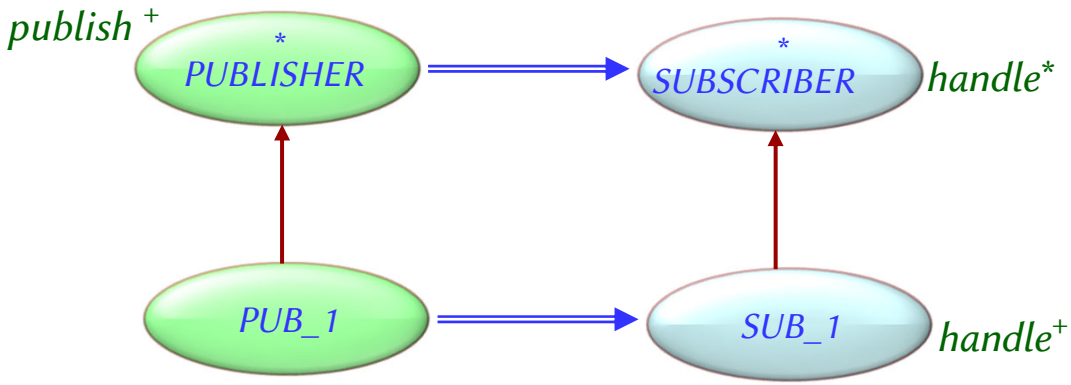




Each descendant of *SUBSCRIBER* defines its own version of *handle*

Publishing an event (2)

The previous loop requires to **ALL items of the list** to call the same procedure...
Do we know a language mechanism allowing to do this more easily?




```
feature publish
        -- Ask all subscribers to
        -- react to current event.
    do
        subscribers.do_all (agent {SUBSCRIBER}. handle (args) )
    end
```

Apply passed procedure to all items of the list

Dynamic binding

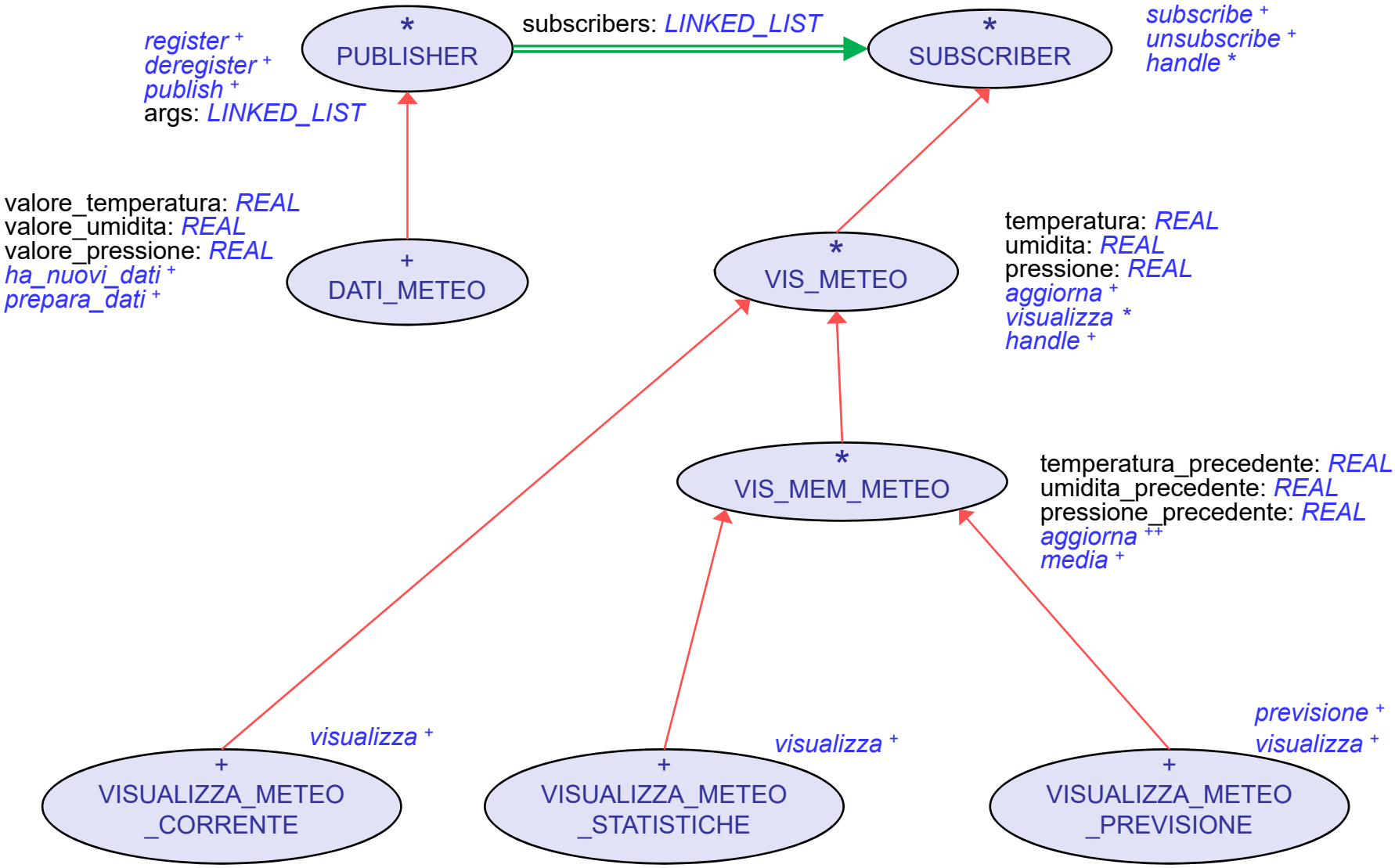
Agent with OPEN TARGET

A practical example

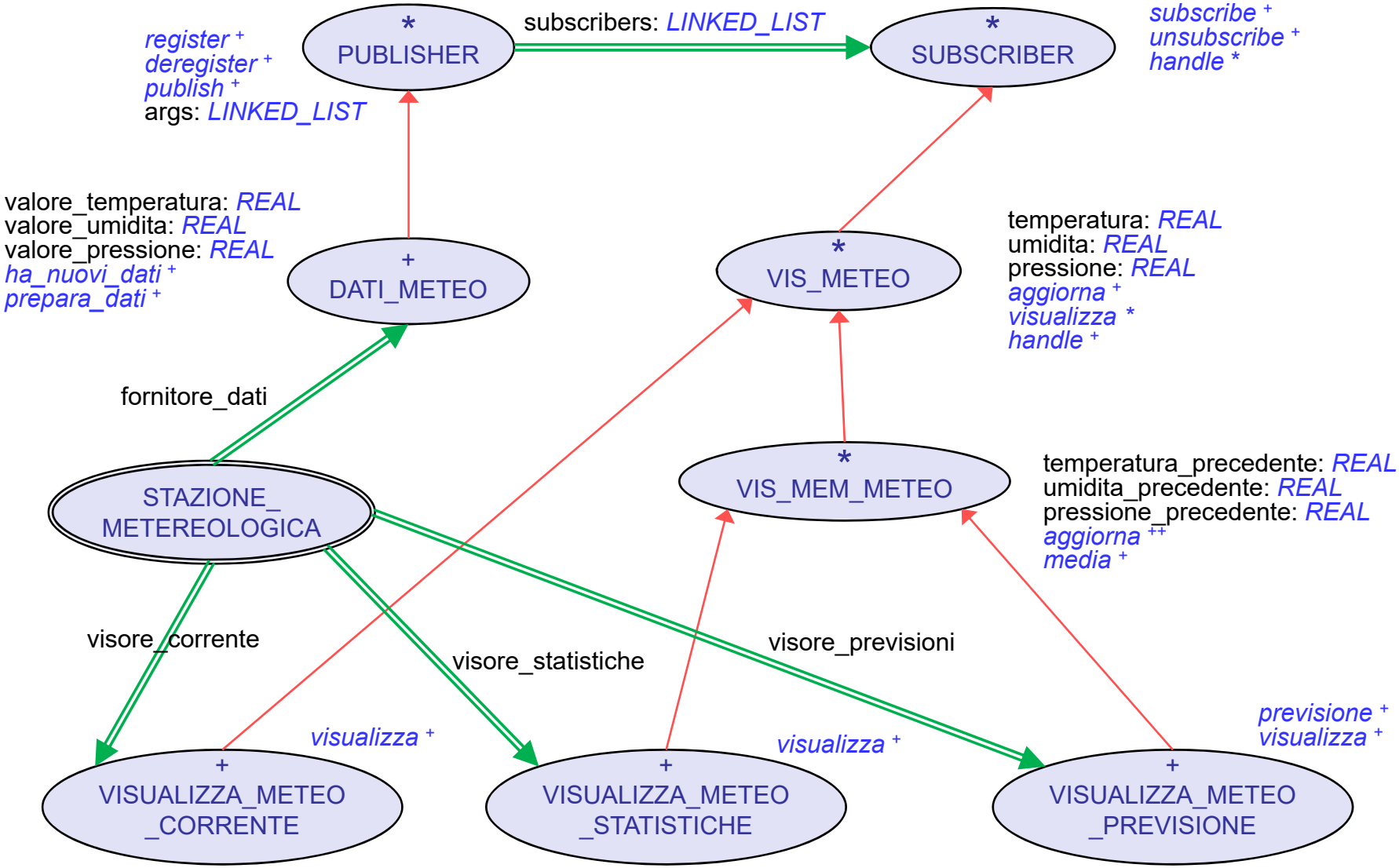
- In a meteorological station sensors of various kinds (temperature, pressure, humidity, ...) publish their values and renderers of various kinds (textual, graphical, ...) display them.

- First look at the diagrams and then see the demo... (stazione_meteo_BASE)

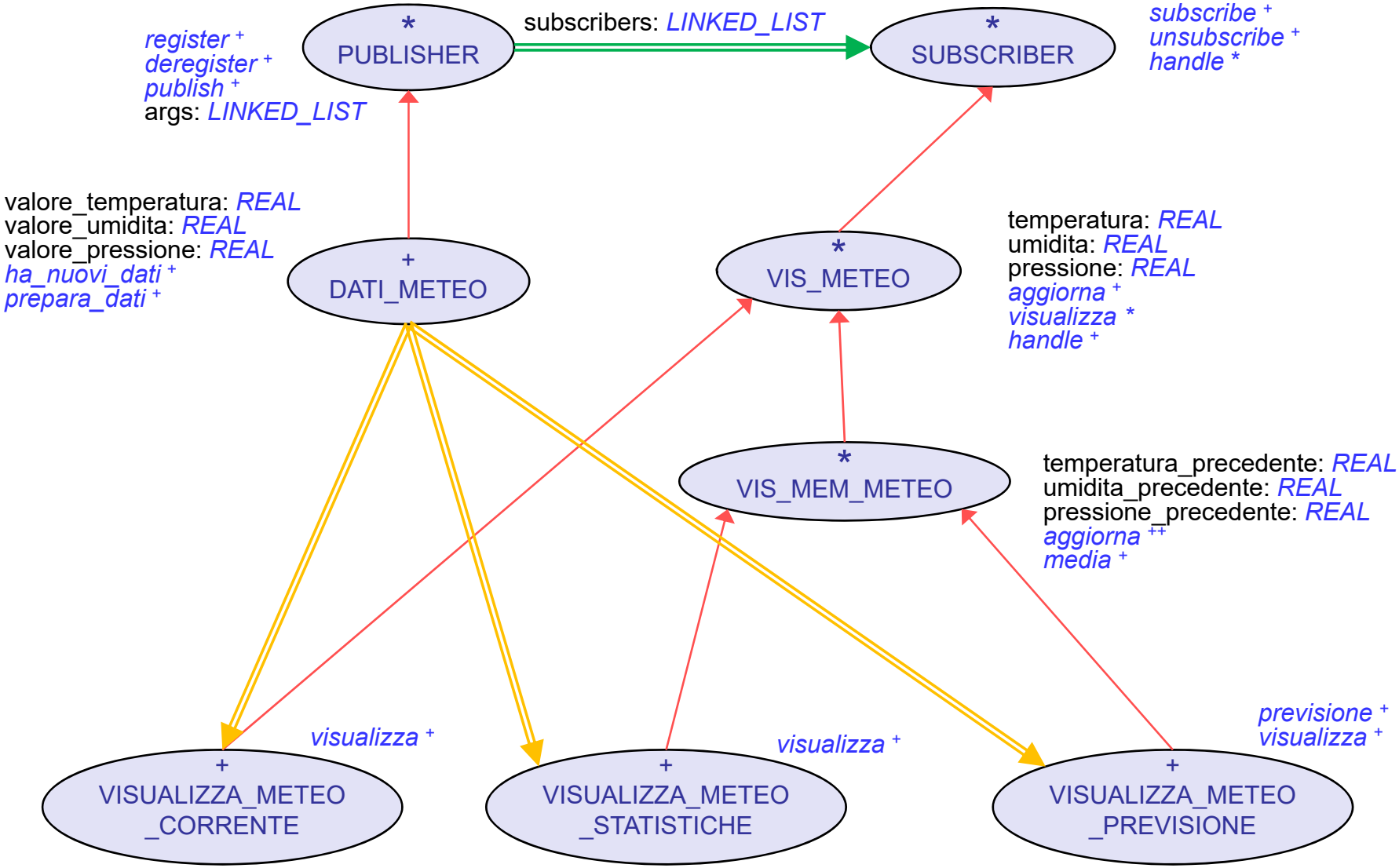
Esempio STAZIONE_METEREOLOGICA (1)



Esempio STAZIONE_METEREOLOGICA (2)



Esempio STAZIONE_METEREOLOGICA (3)



Handling an event with arguments (1)

- In any subscriber class *handle* (*args*: *LIST* [*ANY*]) has to manage arguments

- Assume this means applying a routine defined in the specific subscriber class

```
operation (x: T ; y: U )
        -- x and y are the event's arguments.
    do
        ...
    end
```

- Then implementation of *handle* in the subscriber subclass is like:

```
handle (args: LIST [ANY])
    -- React to publication of one event.
    do
        if args.count >=2 and then
            (attached{T} args.i_th (1) as x) and
            (attached{U} args.i_th (2) as y)
        then operation (x, y)
        else -- do nothing or report error
        end
    end
```

For objects internal to the application, type errors should be detected at compile time, not at run time!

Handling an event with arguments (2)

- An alternative approach is to align definitions of *PUBLISHER* and *SUBSCRIBER* using subclasses introducing specific arguments and their type

- An example for 2 args of type *REAL*

```
class
    PUBLISHER_2_REALS
inherit
    PUBLISHER
redefine
    subscribers, publish, register, deregister
feature
    x, y: REAL
    subscribers: LIST [SUBSCRIBER_2_REALS]
    publish
        do
            subscribers.do_all (agent {SUBSCRIBER_2_REALS }.handle(x, y))
        end
```

New argument names have to be introduced, given the inherited *args* cannot be redefined

The same variation applies for *register* and *deregister* whose arguments have to be redefined as *SUBSCRIBER_2_REALS*

Handling an event with arguments (3)

- Similar variation for subclass of *SUBSCRIBER*

```
class
    SUBSCRIBER_2_REALS
inherit
    SUBSCRIBER
rename
    handle as handle_old
feature
    handle: (x, y: REAL)
        -- React to publication of one event.
    deferred
    end
```

Cannot be redefined and has to be renamed...

...so that a correct version of *handle* with the proper arguments can be introduced

Now the implementation of *handle* in the actual subscriber classes can check their type matches the passed types

Handling an event with arguments (3)

- And in the sub-classed specialized *SUBSCRIBER* class:

```
handle (x : T ; y : U )
    -- React to publication of one event.
    do
        ...
    end
```

- This solution loses in generality:
  - sub-classes of *PUBLISHER* and *SUBSCRIBER* are strictly coupled (if one change the other has to change as well)

See demo... (stazione_meteo_ESPLICITI)

Handling an event with arguments (4)

- A better solution:
- Specify publisher & subscriber as generic and constrained to *TUPLE*
  - *PUBLISHER* [G->TUPLE] e *SUBSCRIBER* [G->TUPLE]
- In *PUBLISHER* there is *args* defined as G
- The instance of generic (*args*) is the argument of *handle* and is pushed (**agent**-ized) to all subscribers through *do_all*
- No need to define specialized sub-classes of *PUBLISHER* and *SUBSCRIBER* m
- Application classes uses the kind of *TUPLE* needed according to the application requirements

Handling an event with arguments (5)

- An example for 2 arguments of type *REAL*

```
class
    MY_PUBLISHER
inherit
    PUBLISHER [REAL, REAL]
redefine
    subscribers, publish, register, deregister
feature
    subscribers: LIST [MY_SUBSCRIBER]
    publish
        do
            args := [x, y]
            subscribers.do_all (agent {MY_SUBSCRIBER }.handle(args))
        end
```

The same variation applies for *register* and *deregister* whose arguments have to be redefined as *MY_SUBSCRIBER*

Handling an event with arguments (6)

- And in the *MY_SUBSCRIBER* class:

```
handle (args : TUPLE [arg_1: REAL; arg_2: REAL])
    -- React to publication of one event.
    do
        ...
    end
```

- Now there is no need for specialized (and strictly coupled) subclasses of *PUBLISHER* and *SUBSCRIBER* while the actual publishers *MY_PUBLISHER* and subscribers *MY_SUBSCRIBER* are able to detect a mismatch between expected arguments and passed arguments

See demo... (stazione-meteo-argomenti-TUPLE)

Observer pattern problems

1. Handling of arguments requires special care

2. A subscriber can react to only one kind of event from one publisher (there is only one *handle*)

3. Subscribers have to know publishers although they are interested only in events

4. It is not easy to extend already existing software to use this pattern, above at all in languages without multiple inheritance

Choosing the right abstraction

The only significant feature of publisher is *publishing event of a given type*

The only significant feature of subscriber is *subscribing event of a given type*

The features of an event are:

- commands to publish and to subscribe
- the notion of argument(s)

Then the right abstraction is

- Only one generic class: *EVENT_TYPE*
- Two features: *publish* and *subscribe*
- Using agents in subscribing

For example: A map widget *Paris_map* that reacts in the way defined in *find_station* when clicked (event *left_click*):

EVENT_TYPE class

The only needed class is *EVENT_TYPE* [*G* –>*TUPLE*]

On the publisher side, e.g. a GUI library or a data producer:

- declare an entity modeling the event, an instance of *EVENT_TYPE*:

  click : *EVENT_TYPE* [*INTEGER, INTEGER*]

- create (just once) the instance:

Remember the old syntax with **TUPLE[...]**

On the subscriber side, e.g. an application or a data consumer:

- subscribe only once, to act or to receive notification from the instance (i.e.: receive information about the fact that something has happened)

  click.*subscribe* (**agent** *find_station*)

On the publisher side, e.g. a GUI library or a data producer:

- alert subscribers whenever needed (i.e.: spread information about something that has happened related to that type of event):

  click.*publish* (*x_coordinate, y_coordinate*)

Advantages

1. Each event has its specific signature, the actual parameters for

   EVENT_TYPE [G –> TUPLE]

2. Each subscription specifies the action to be taken, passed as agent, allowing to reuse existing model features
3. Complete decoupling of publishers and subscribers
4. Each class can define its own event type. Then publishers can publish the event and pass with it arguments related to their specific context, e.g.
   - for a button, its name
   - for a window, its size
   - for a field, its content

Towards implementation

Which class declares features of type *EVENT_TYPE*?

A class interested in publishing events, that is:

- A general class of your application, if events are of general interest, independently from the context they occur within
- A specific class (e.g. the one modeling buttons, or menus, or ...) if events are of interest in a specific context (i.e., the button, the menu, ...)

EVENT_TYPE features (1)

Each instance of *EVENT_TYPE* is a list of **agent**-ized procedures:

class

EVENT_TYPE [*EVENT_DATA* –> *TUPLE*]

inherit

LINKED_LIST [*PROCEDURE* [*EVENT_DATA*]]

Ensure passing mismatched arguments is caught at compile time

Subscribers register their actions through this feature

feature

subscribe (*action*: *PROCEDURE* [*EVENT_DATA*])

-- Register *action* to be executed for this event

do

extend (*action*)

ensure

present : *has* (*action*)

end

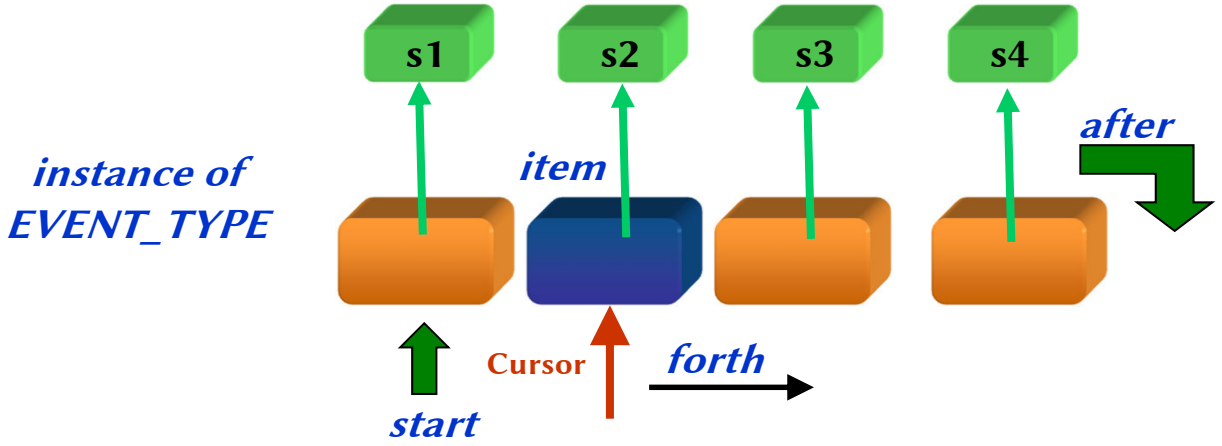
EVENT_TYPE features (2)

```
feature publish (args)
        -- Execute all actions
        -- subscribed to current event.
    do
        from
            start
        until
            after
        loop
            item.call(args)
            forth
        end
    end
```

Must be a tuple

or the equivalent **across** form

Dynamically execute the right procedure



Example using *EVENT_TYPE*

The publisher defines a feature returning an instance of *EVENT_TYPE*:

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

The body is executed only once. Subsequent times the previous result is returned (function) or nothing happens (procedure)

The subscribers register actions to be executed to react to events:

```
Paris_map.left_click.register (agent find_station)
```

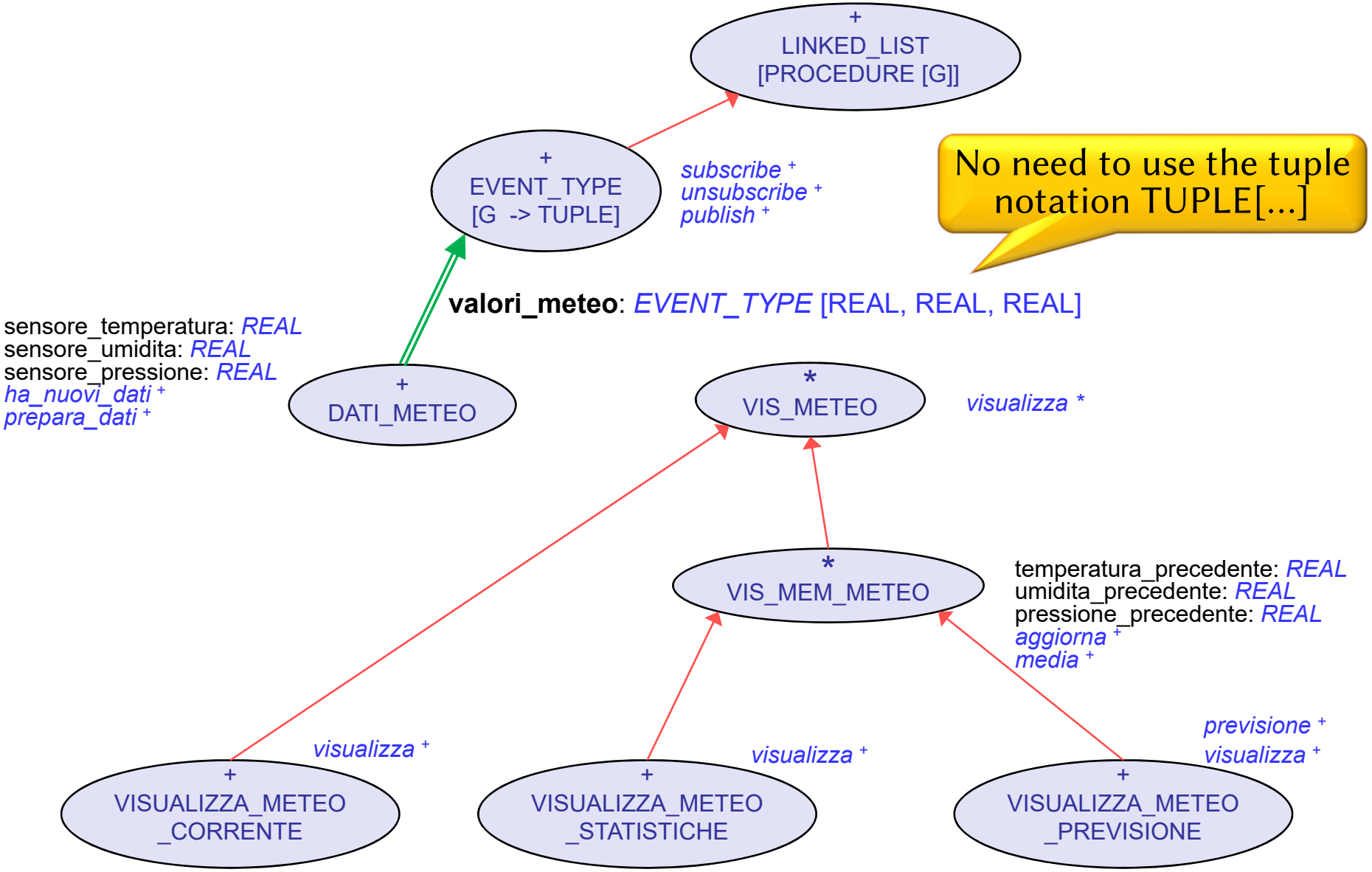
The instance of *EVENT_TYPE* gets created the first time a subscriber calls it

The publisher publishes the specific event (the publisher has received from the hardware *x_position* and *y_position* through system software):

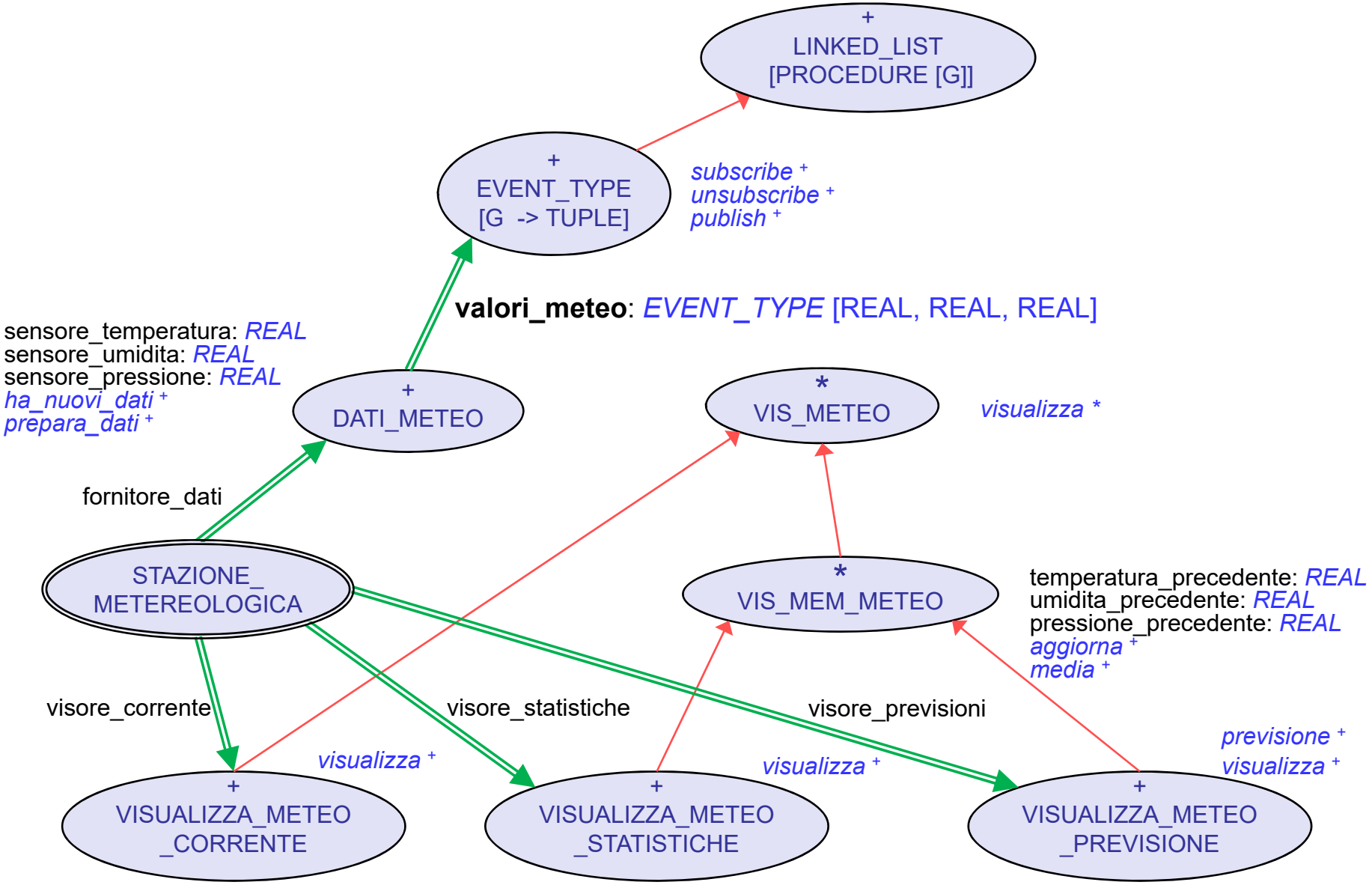
```
left_click.publish (x_position, y_position)
```

Remember the old syntax with [...]

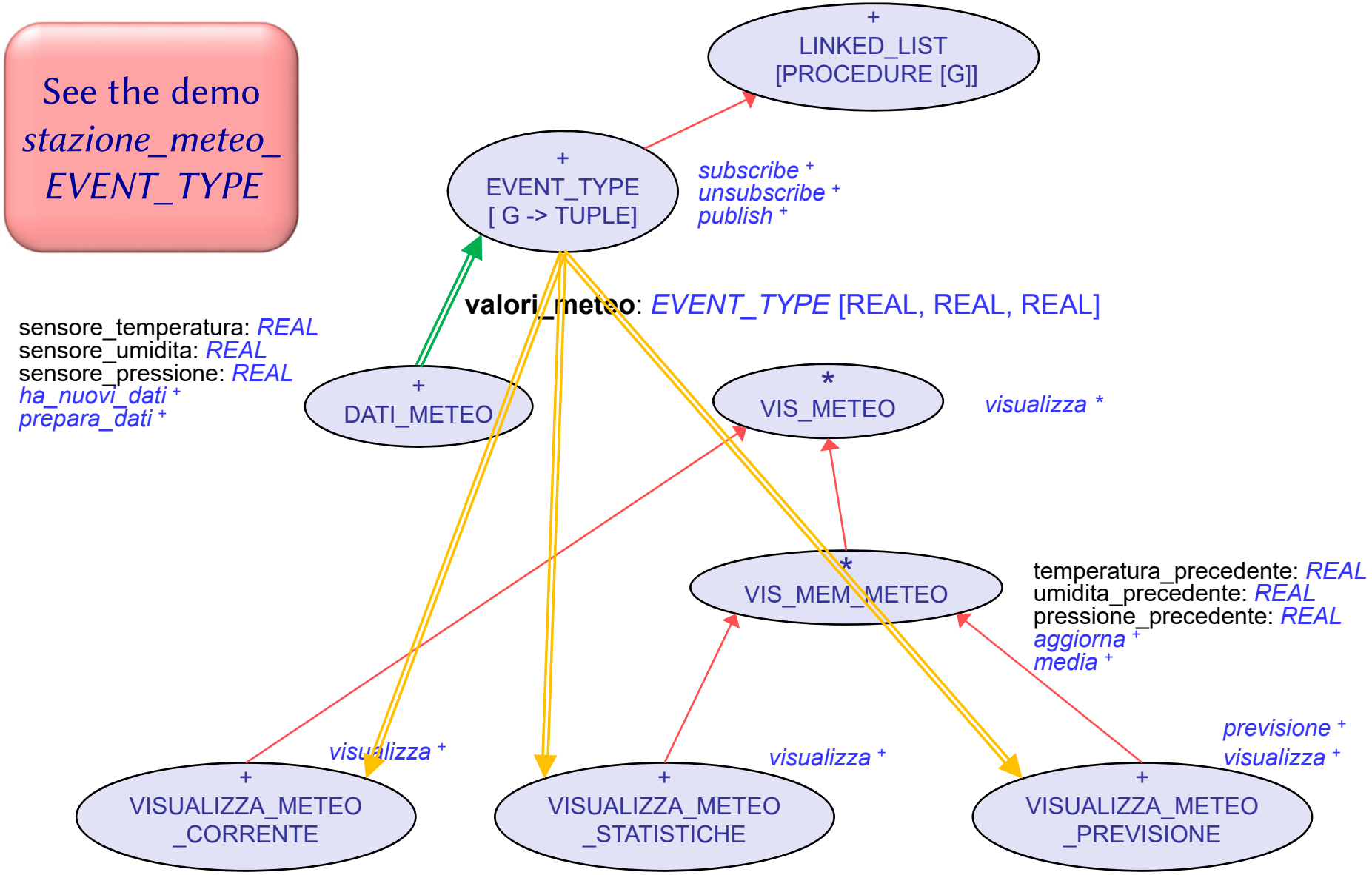
Esempio STAZIONE_METEREOLOGICA (1)



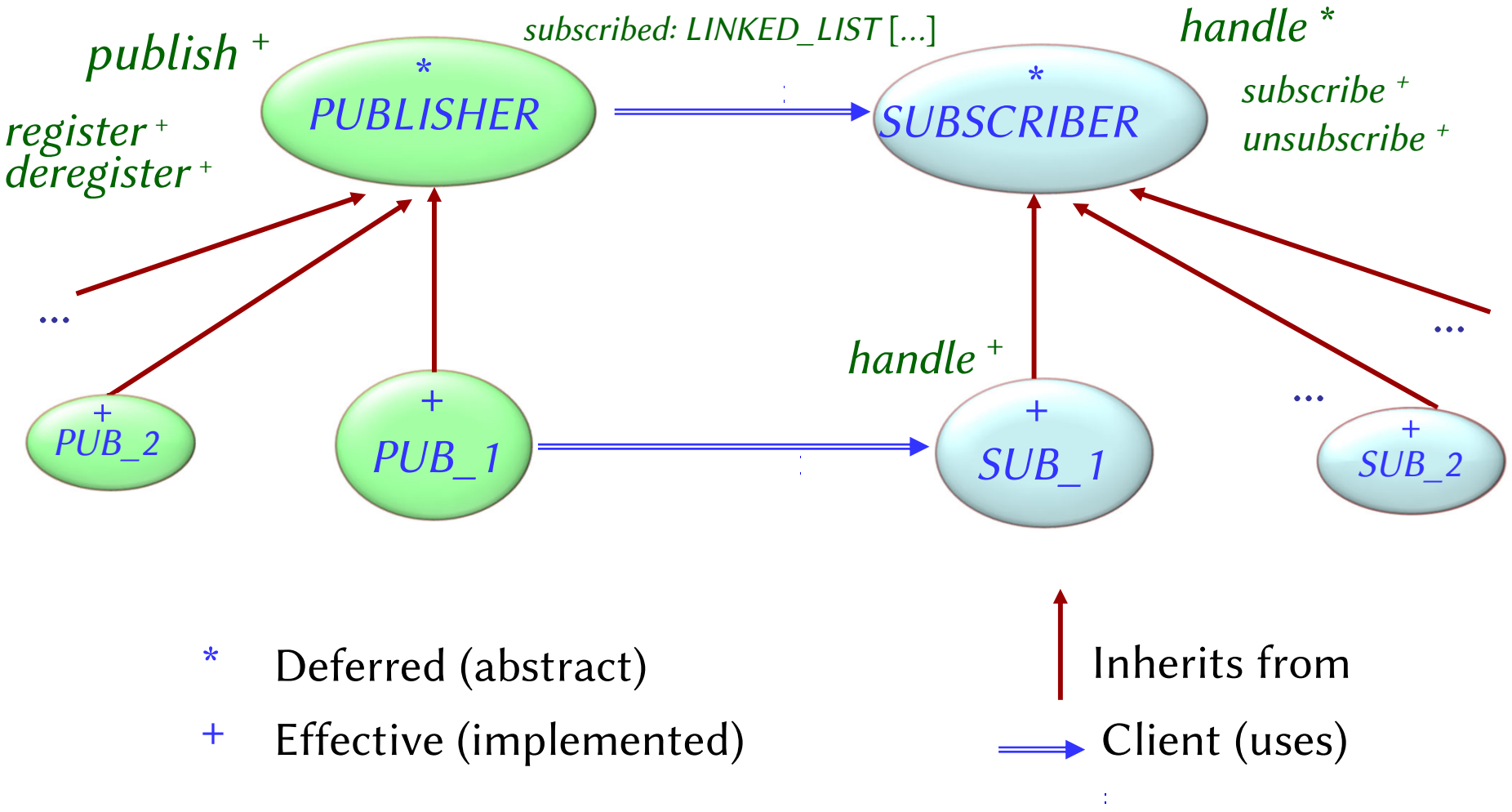
Esempio STAZIONE_METEREOLOGICA (2)



Esempio STAZIONE_METEREOLOGICA (3)



Reminder: the Observer Pattern



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 *handle* to call appropriate model routine

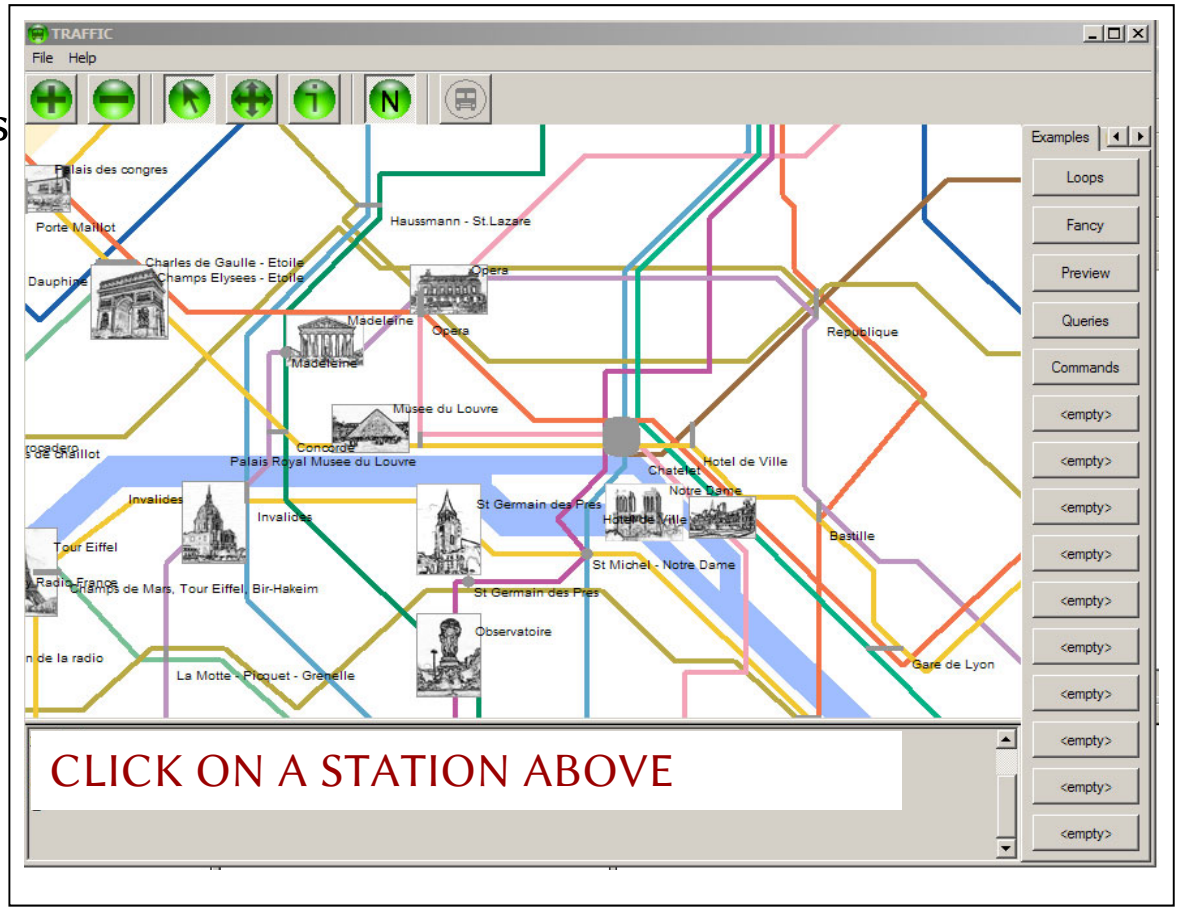
- With the Event Library:
  - No new classes (use library classes directly)
  - No need of writing *handle*s: can reuse the existing model routines directly as agents

Graphical User Interface

Specify that when a user clicks on a station the system must execute

find_station (*x*, *y*)

where *x* and *y* are the mouse coordinates and *find_station* is a specific procedure of your system.



Important issues for interactive applications

1. Keeping the Business Model and the User Interface separate
   - Business model (or just *model*): core functionality of the application
   - User Interface: manage interaction with users

2. Minimizing "glue code" between the two

3. Preserving the ability to reason about programs and predict their behavior

MODEL – VIEW separation

In a given software application

- **The model** (or business model) represents and processes data of the application domain
  - The true purpose of the application
- **A view** is a presentation of (a part of) application domain data to the outside (persons, software, devices), for input/output
  - The interaction with the rest of the world
  - Many views depending on the kind of interacting entities
- Manage the two aspects with two **minimally coupled** different parts of the application

Model-View and Publisher-Subscriber

These are orthogonal concepts

Both publishers and subscribers may interact with the model as well as with the views

Example for a text processing application:

- A publisher may publish an event as a consequence of something happening
  - in the view (user clicking a button)
  - or in the model (spell checker flagging a word)

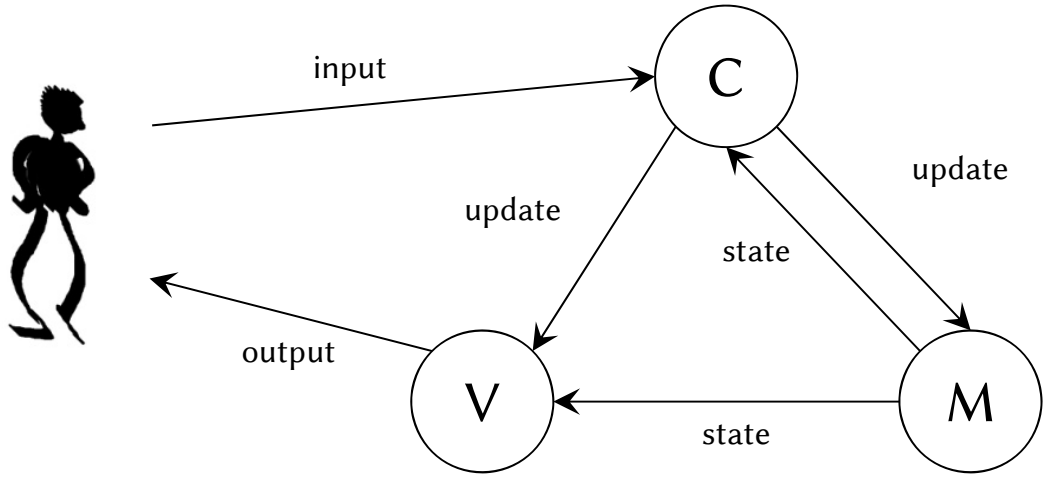
- A subscriber handling an event can affect
  - the view (updating screen after a word deletion)
  - or the model (updating after a word deletion)

Architecture MVC (Model-View-Controller)

View (*vista*): produces output, depending on the current state of the model

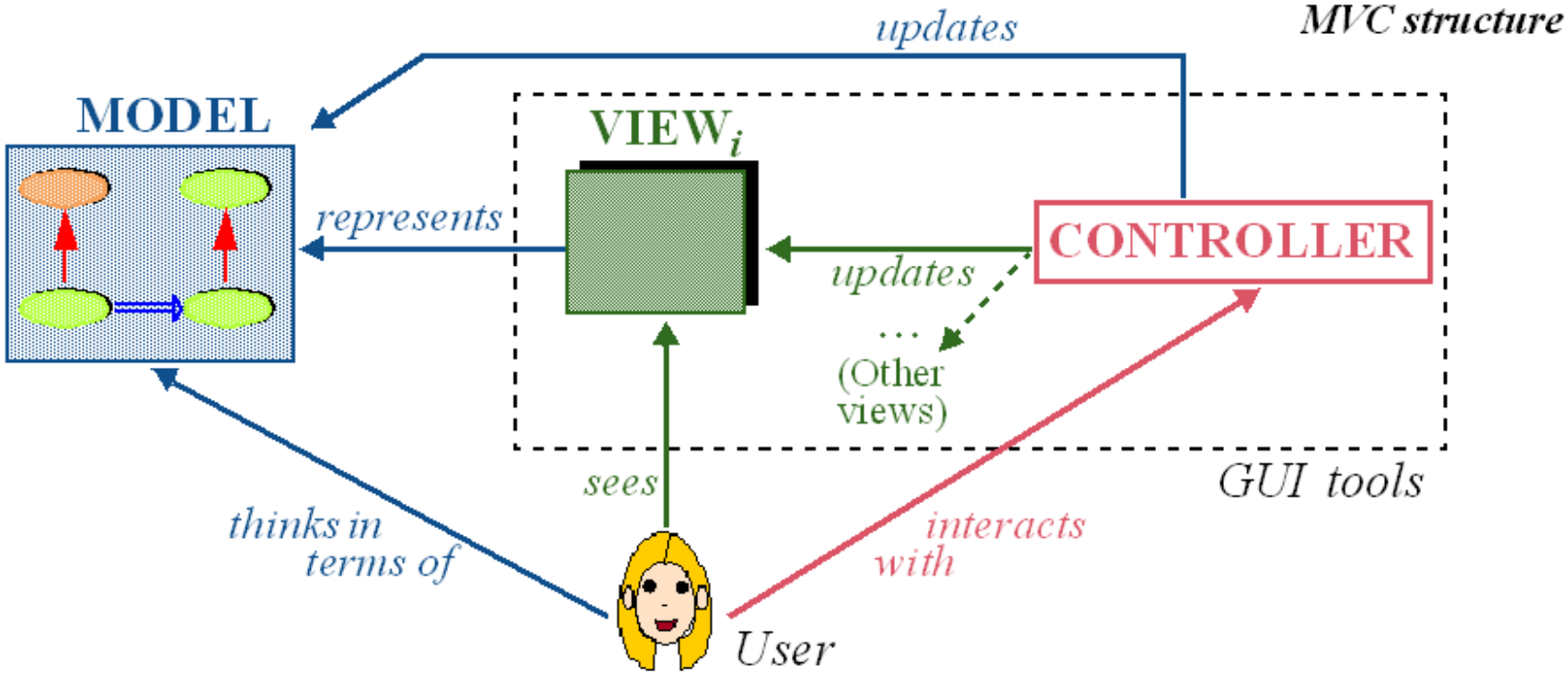
Controller (*controllore*): receives input and manages changes to the view and the model (driving them as needed)

Model (*modello*): manages data in the application domain and their evolution



Model-View Controller

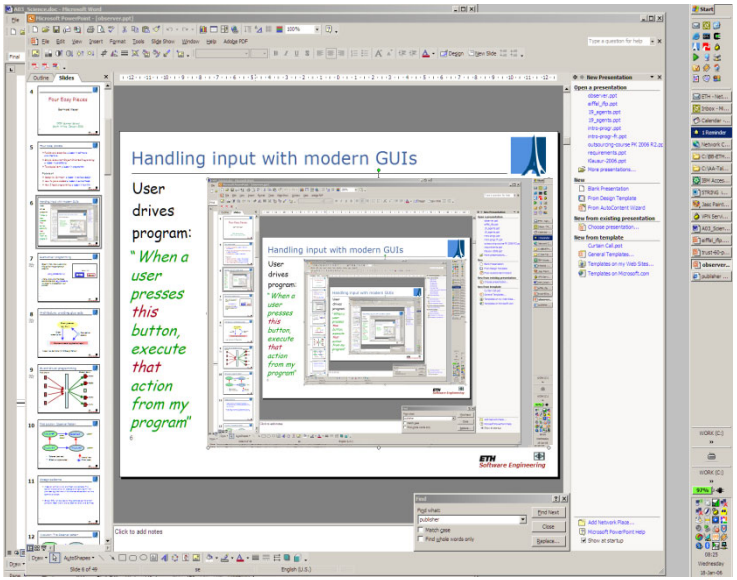
(Trygve Reenskaug, 1979)



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

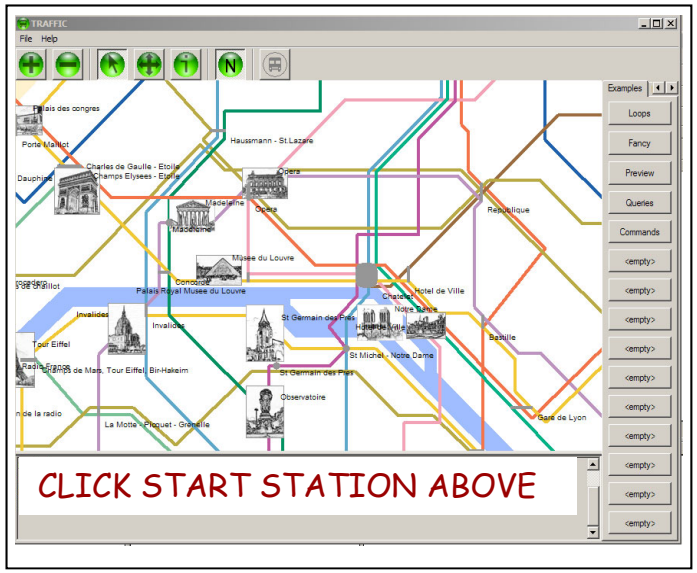
More precisely: Event_type - Context - Action Table

Event type	Context	Action
Left_click	Save_button	*Save_file*
Left_click	Cancel_button	*Reset*
Left_click	Map	*Find_station*
Left_click	…	…
Right_click	…	*Display_Menu*
…		…

Action-event table

Set of triples

[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.