ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Lecture 17: Event-driven programming and Agents

(Two-part lecture, second half next week)

Our goal for this (double) lecture

We will extend our control structures with a more flexible mechanism where control is decentralized

Resulting mechanism: **agents** (Eiffel); other languages have **delegates** (C#), **closures** (functional languages)

Applications include:

- Interactive, graphical programming (GUI)
  (Our basic example)
- Iteration
- Numerical programming
- Concurrency

Handling input through traditional techniques

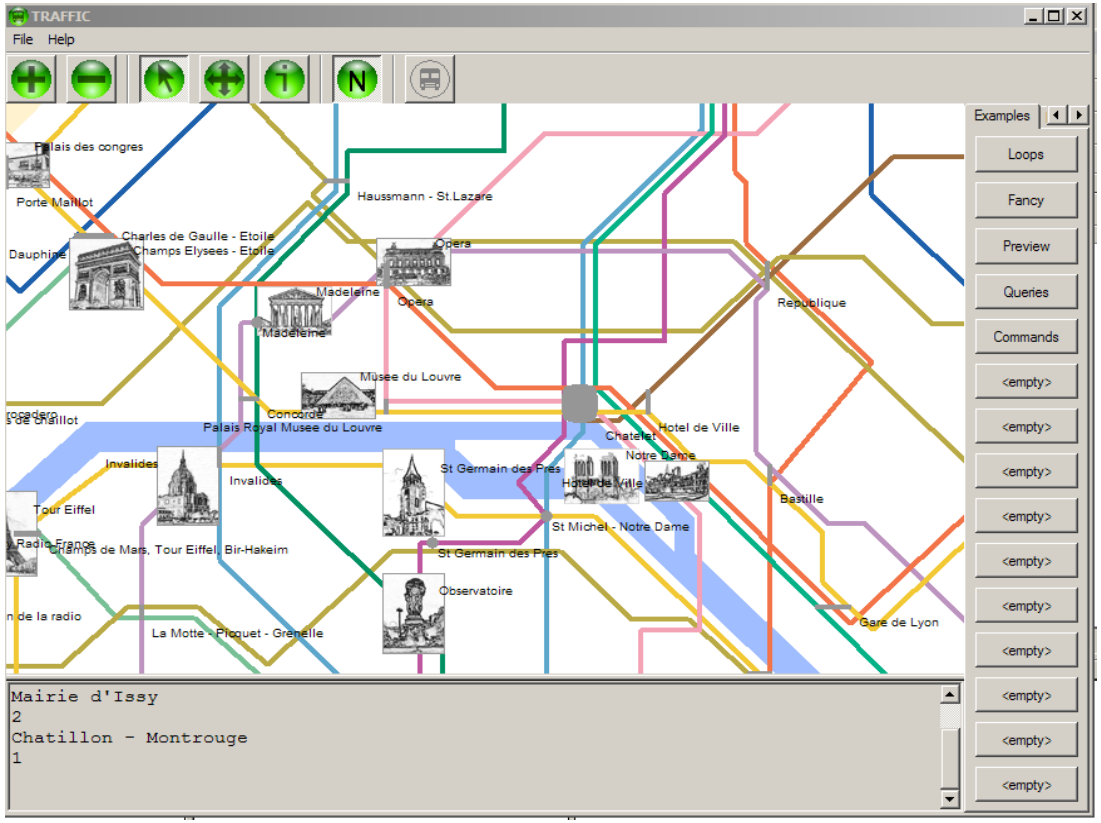
Program drives user:

```
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"

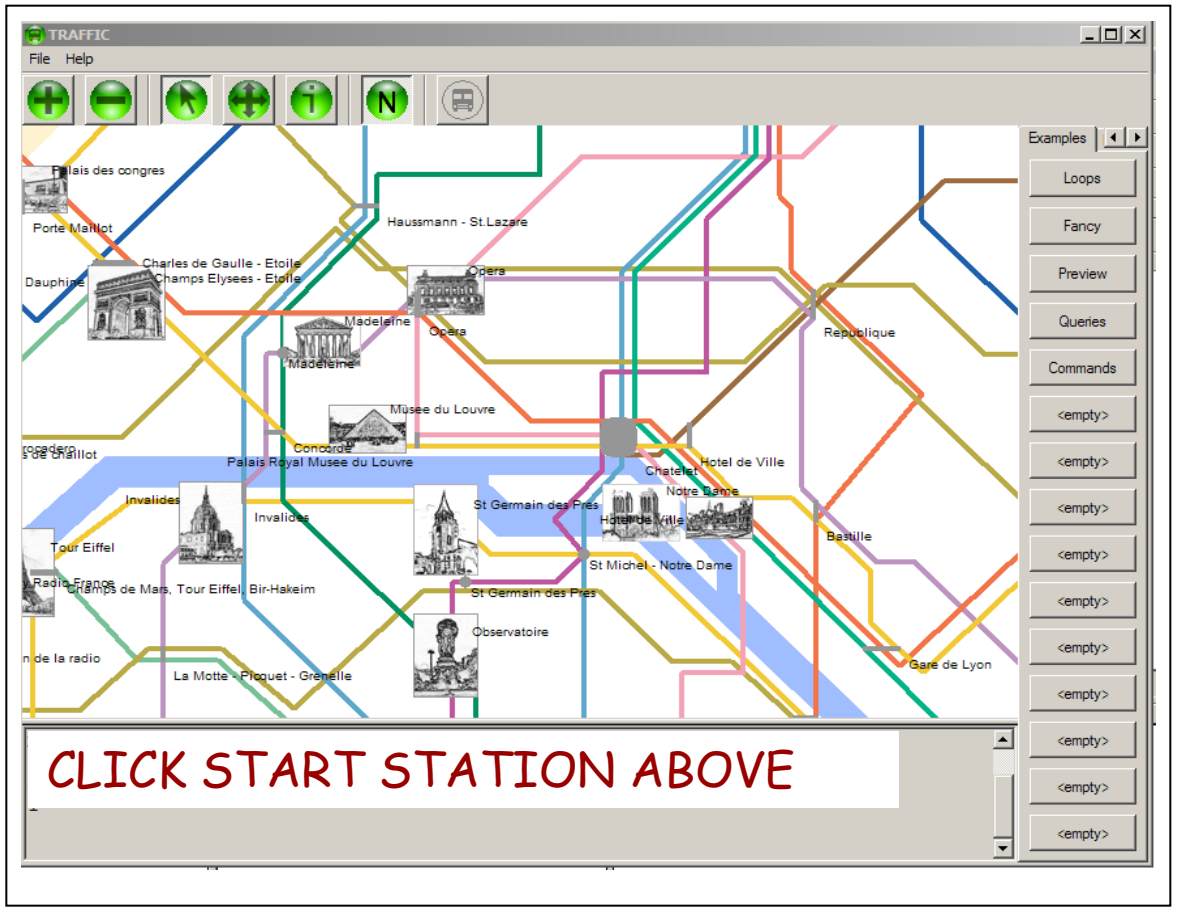


Event-driven programming: an example

Specify that when a user clicks this button 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.



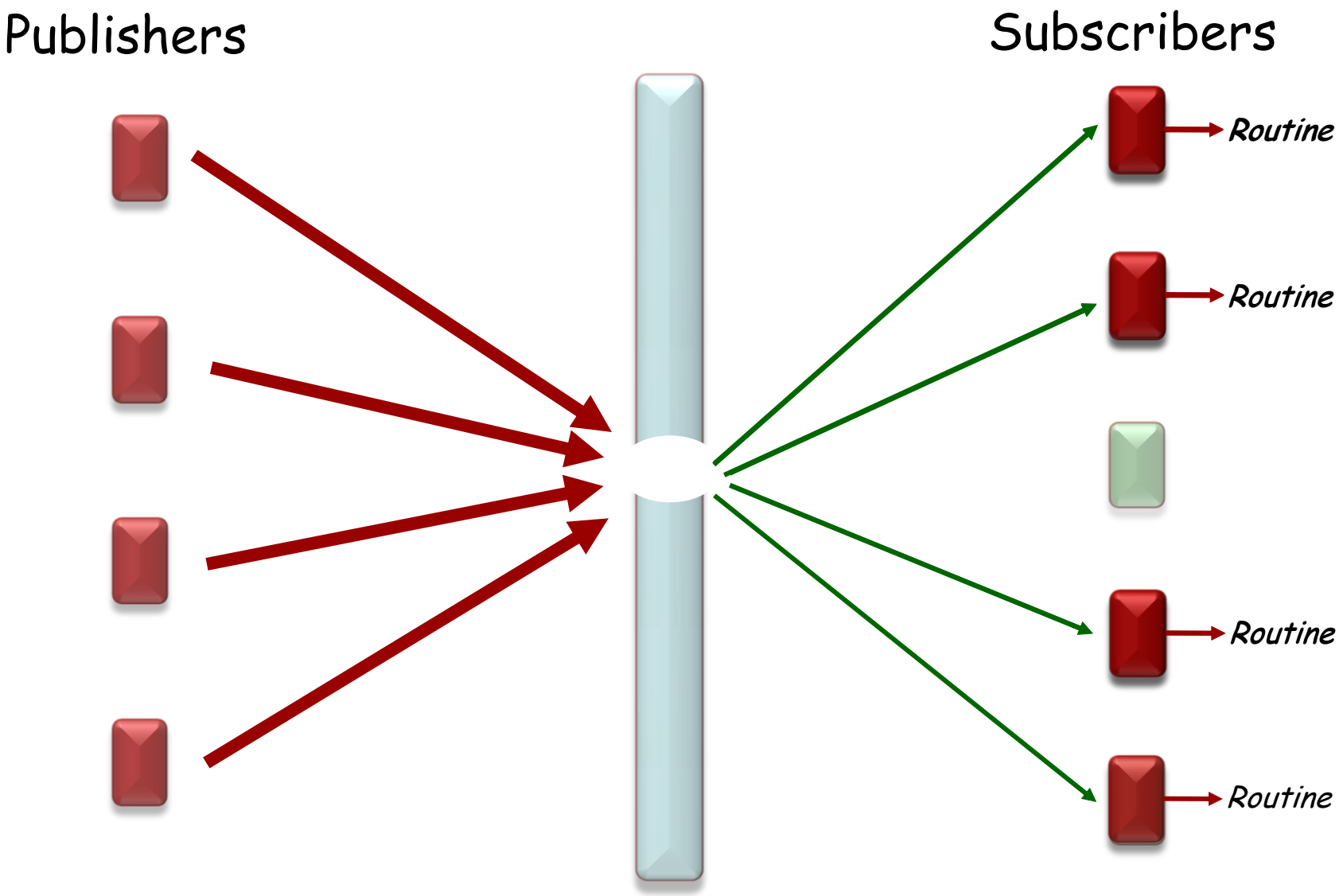
Some issues

1. Keeping the "business model" and the UI separate
   - Business model (or just *model*): core functionality of the application
   - UI: interaction with users

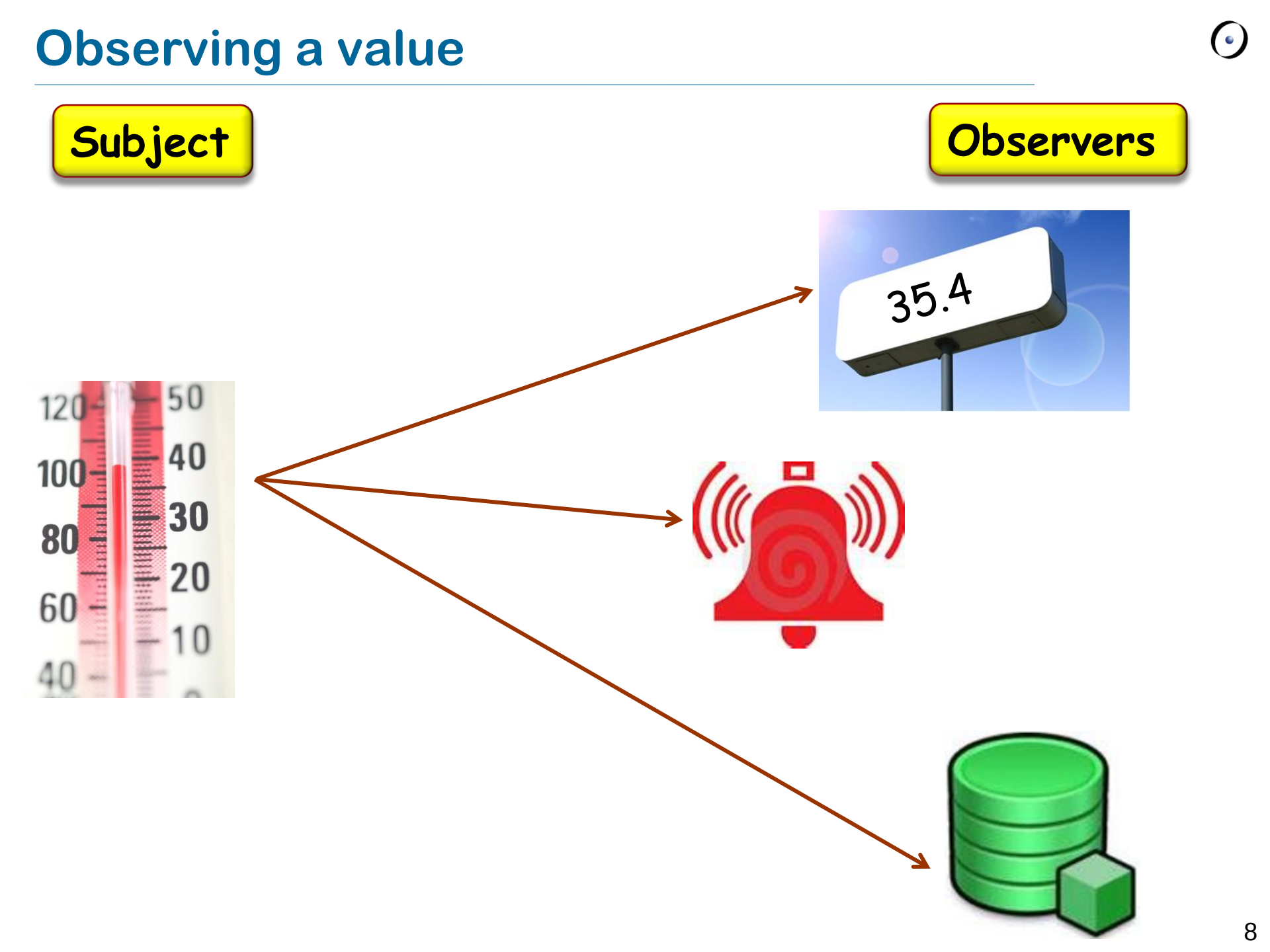
2. Minimizing "glue code" between the two

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

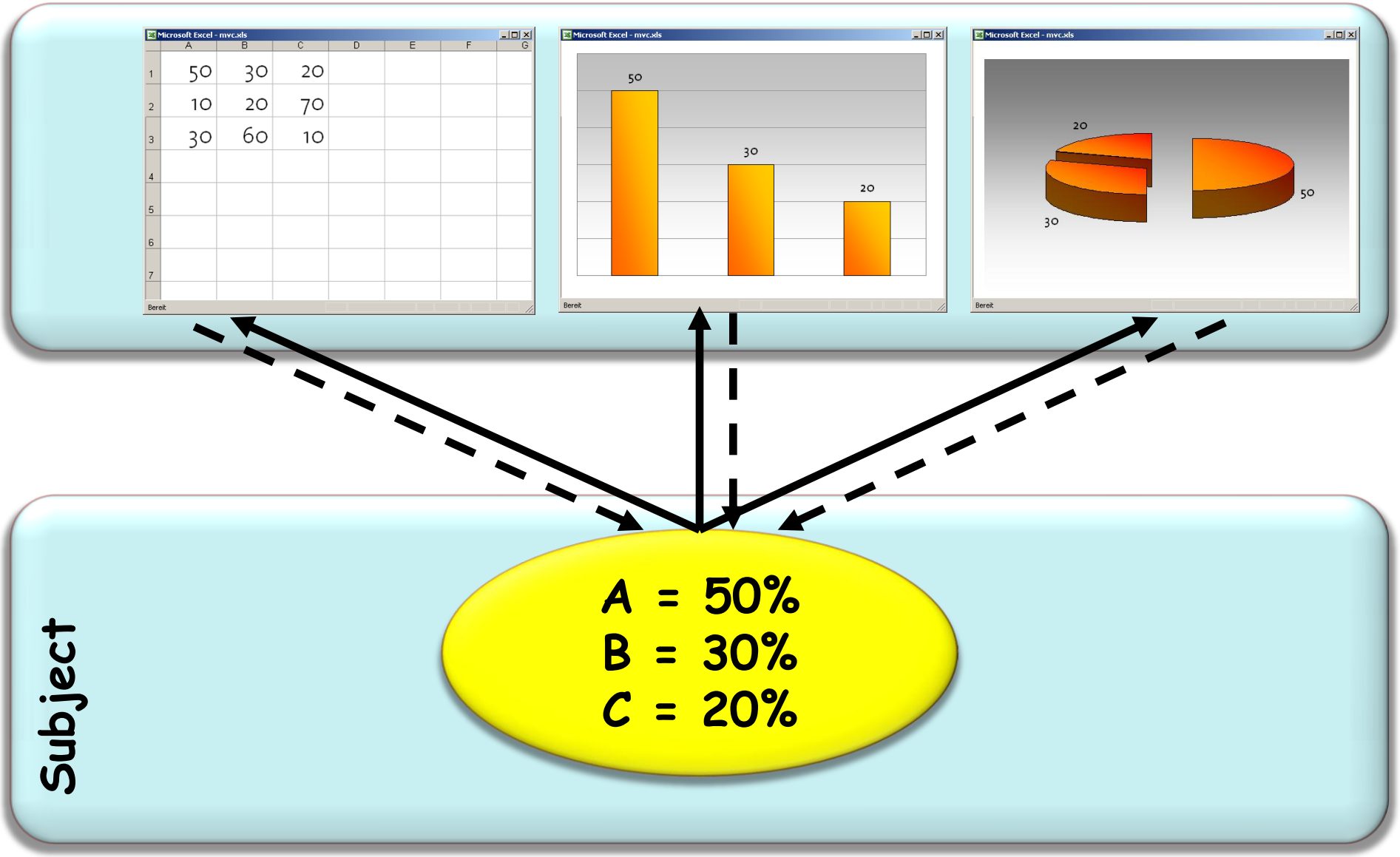
Event-driven programming: a metaphor



Observing a value



Observing a value



Alternative terminologies

Observed / Observer

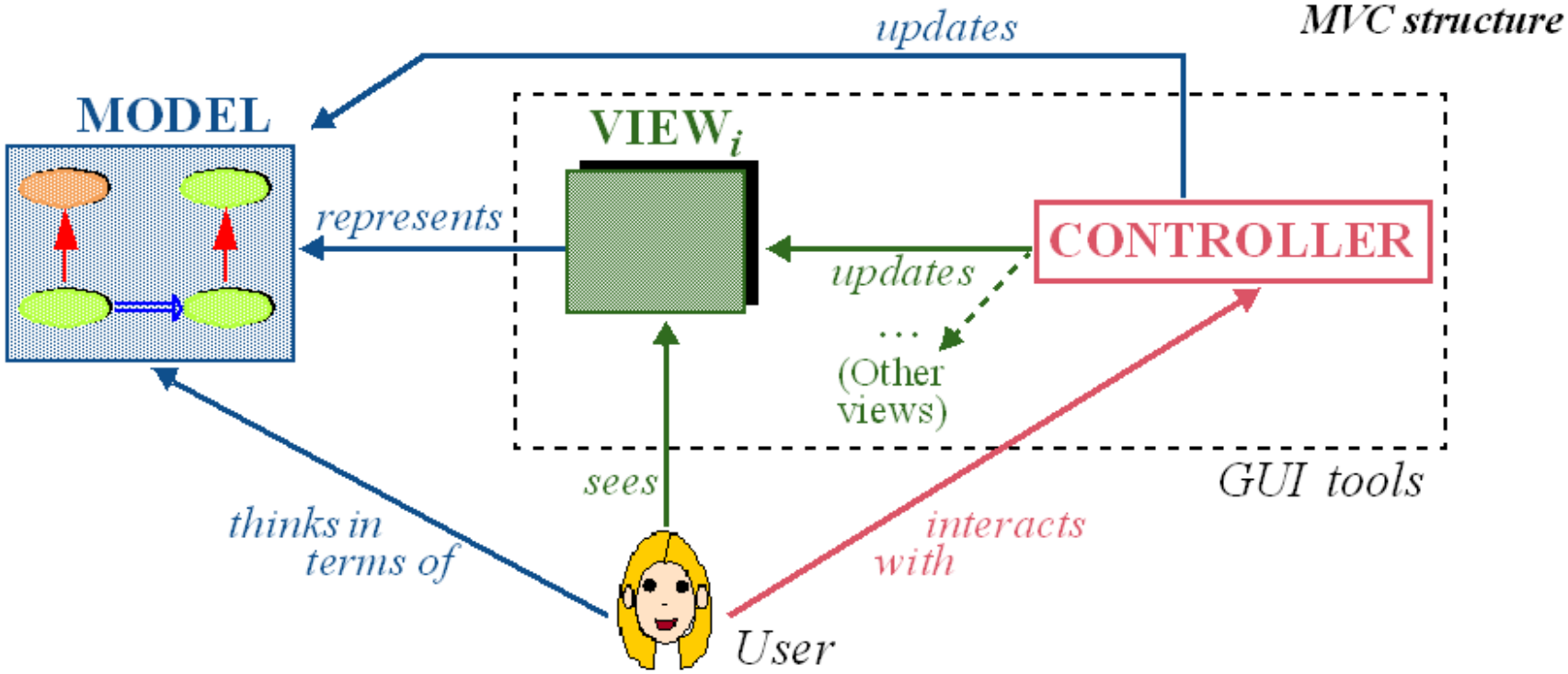
Subject / Observer

Publisher / Subscriber

In this presentation:
Publisher and Subscriber

Model-View Controller

(Trygve Reenskaug, 1979)

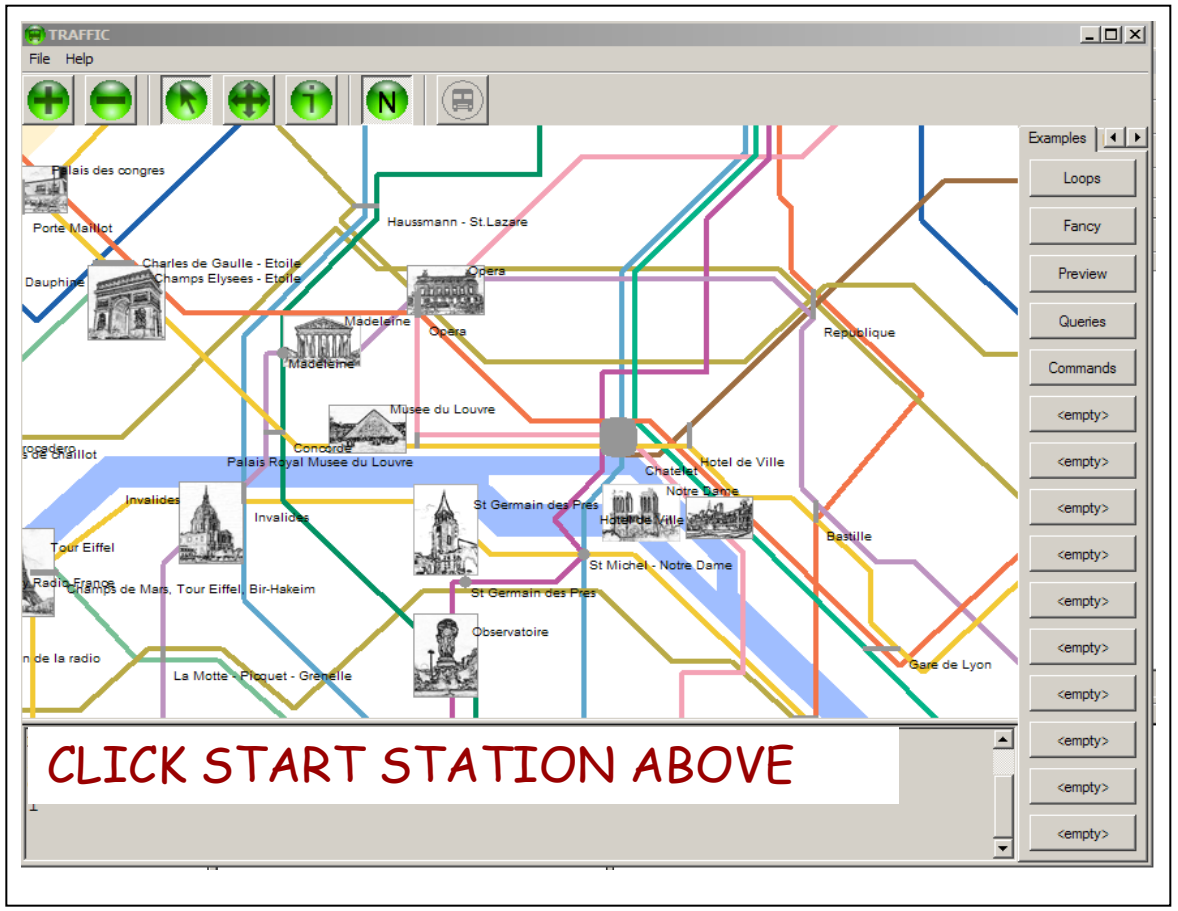


Our example

Specify that when a user clicks this button 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.



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 EventHandler delegate and the EventArgs base class.

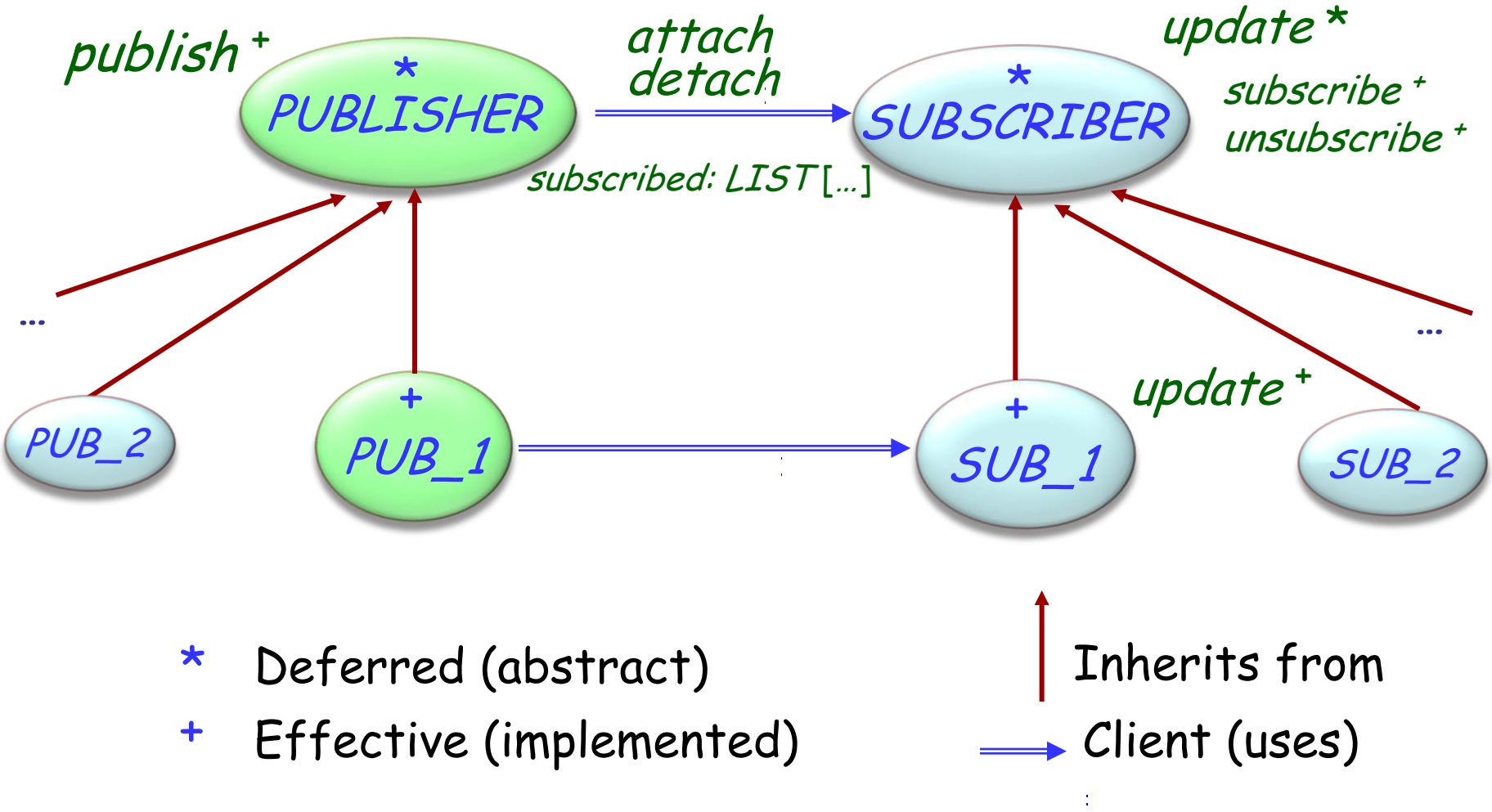
Event arguments

Some events are characterized just by the fact of their occurrence

Others have arguments:

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

An architectural solution: the Observer Pattern

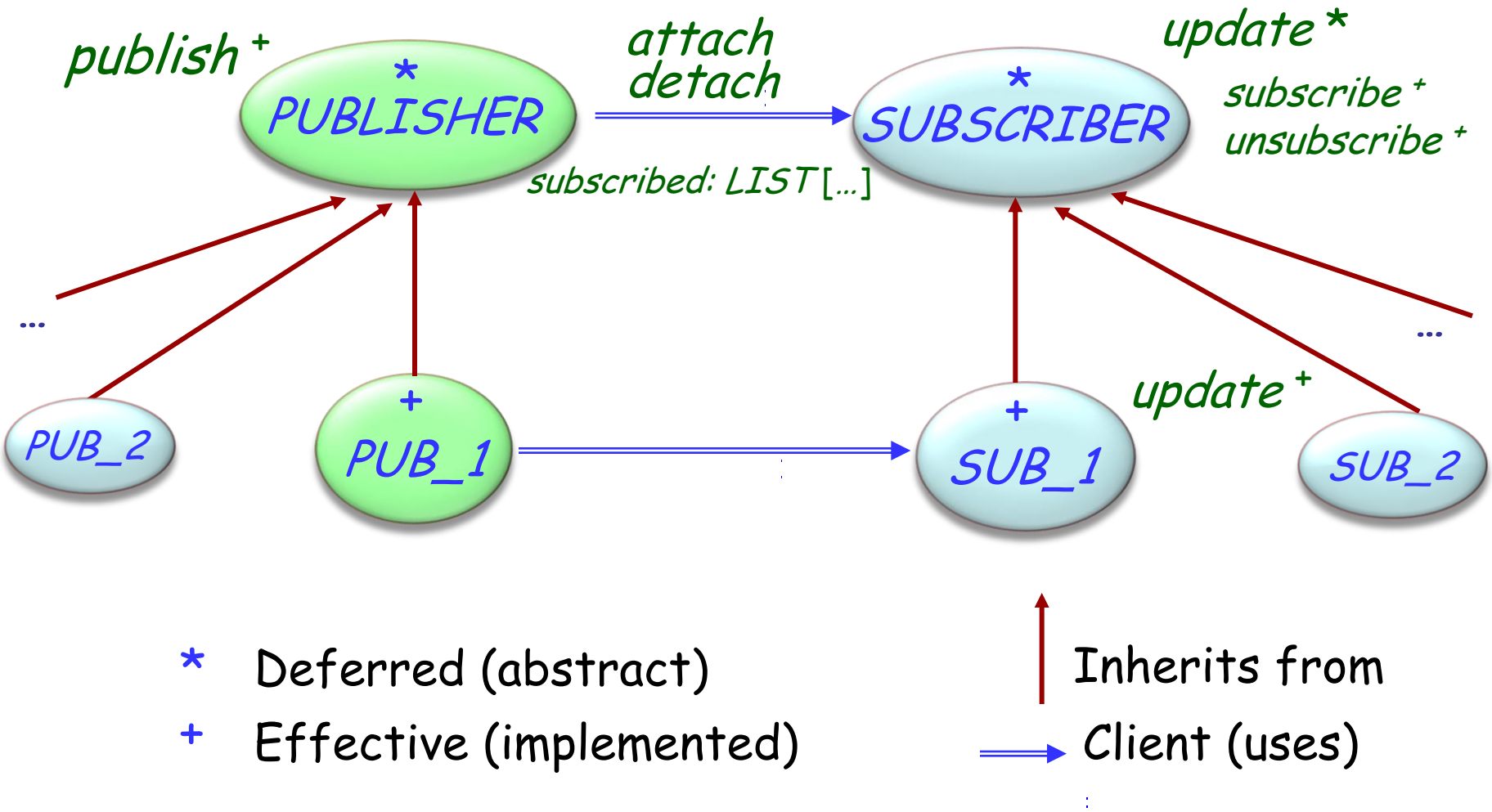


Design patterns

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

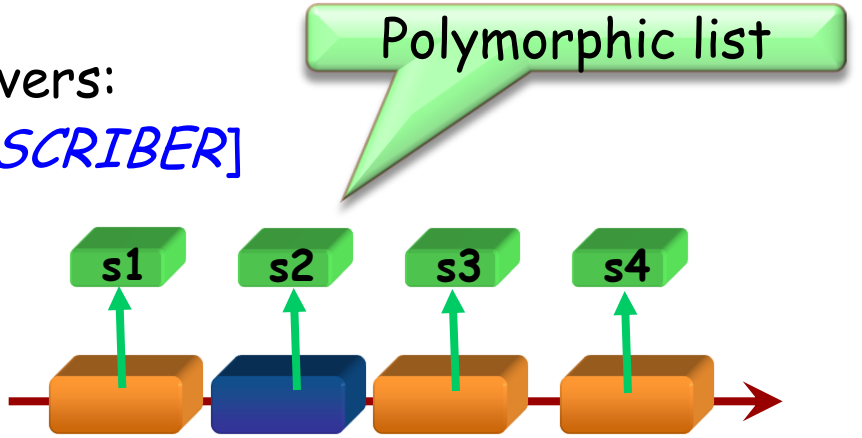
A solution: the Observer Pattern



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*:

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

Attaching an observer

In class *PUBLISHER*:

Why?

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

                subscribed.extend (s)
            end
```

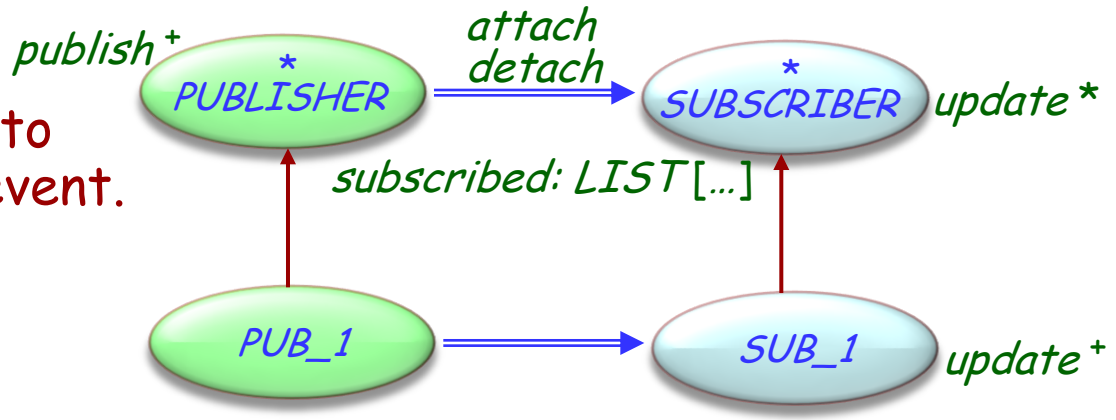
Note that the invariant of *PUBLISHER* includes the clause

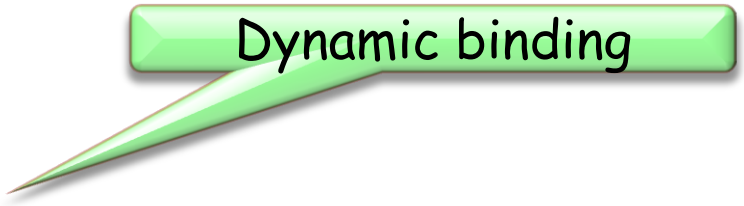
subscribed /= Void

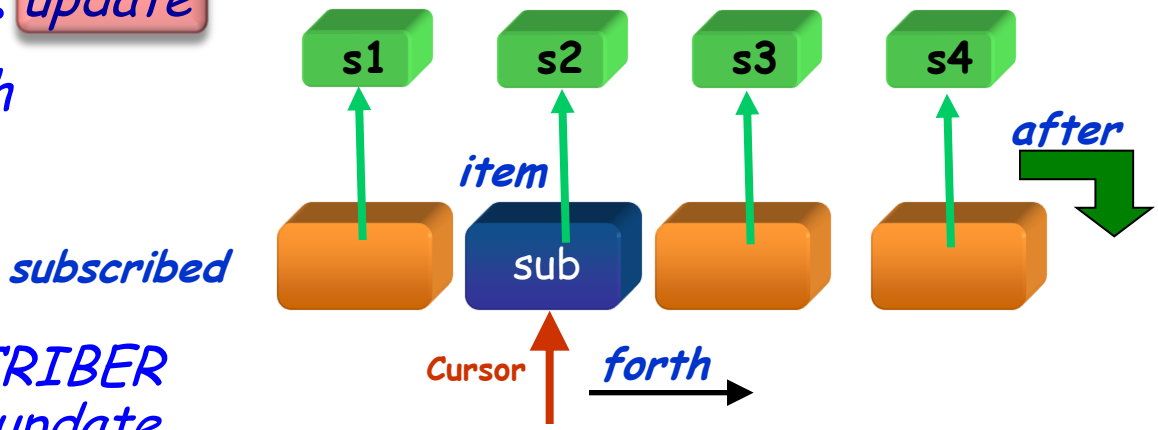
(List *subscribed* is created by creation procedures of *PUBLISHER*)

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
```





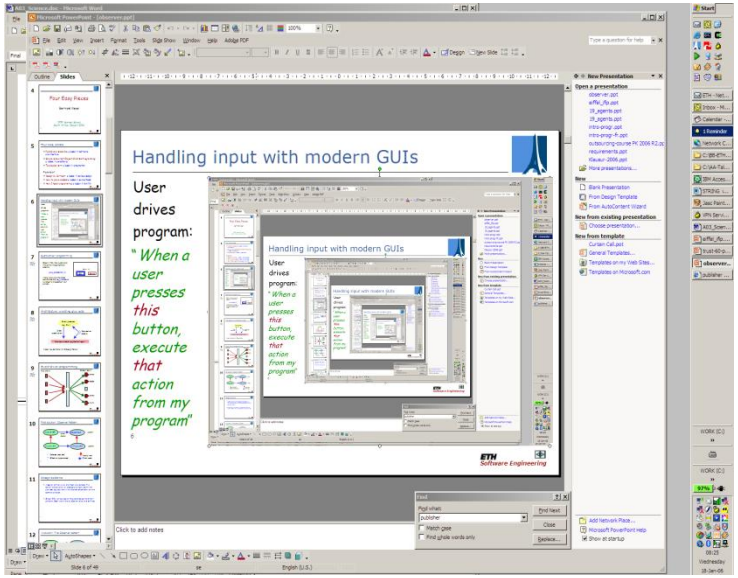


Each descendant of *SUBSCRIBER* defines its own version of *update*

Observer pattern (in basic form)

- Publisher objects know about subscribers
- Subscriber classes (and objects) know about their publishers
- A subscriber may subscribe to at most one publisher
- It may subscribe at most one operation
- Handling of arguments (not detailed in previous slides) requires special care
- The solution is not reusable: it 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

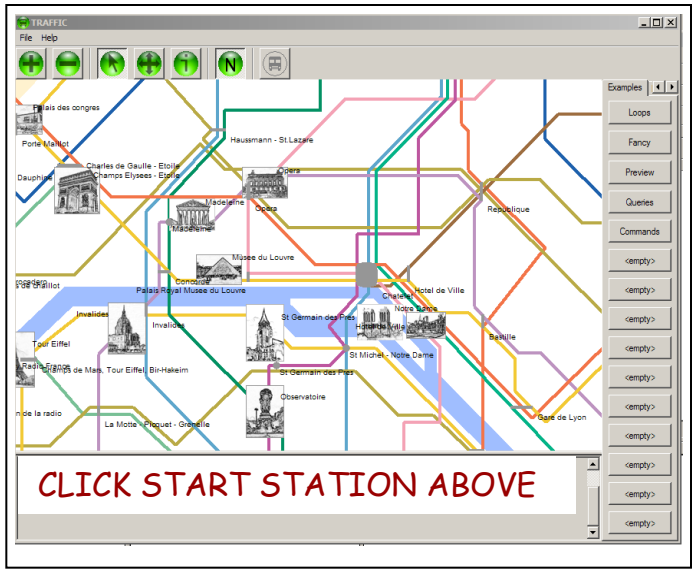
More precisely: Event_type - Context – Action Table

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.

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

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.)

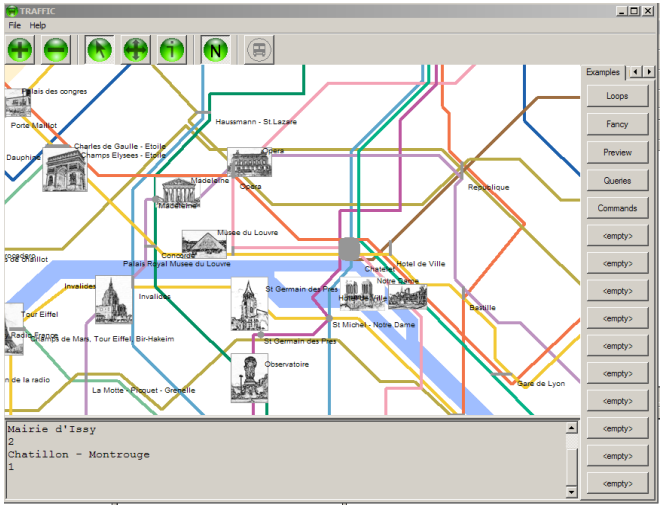
Agents (or delegates in C#) provide a higher-level, more abstract and safer technique by wrapping the routine into an object with all the associated properties.

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

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:

```
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_positition, y_position])
```

The subscribers ("observers") subscribe to events:

```
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

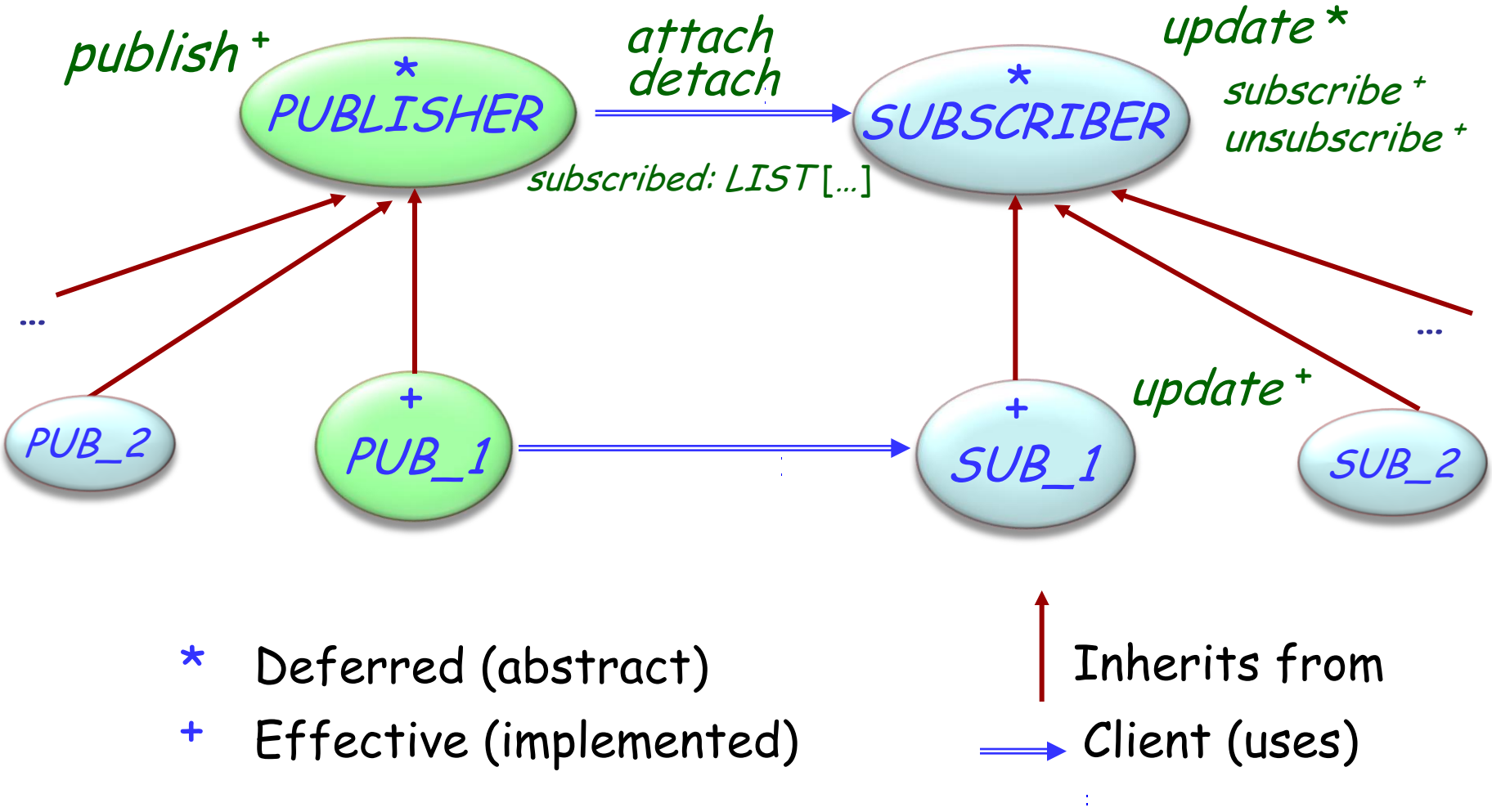
In case of an existing class *MY_CLASS*:

- With the Observer pattern:
  - Need to write a descendant of *SUBSCRIBER* and *MY_CLASS*

    May lead to useless multiplication of classes
  - Effect *update* to call appropriate model routine

- With the Event Library:
  - No new classes (use library classes directly)
  - Can reuse the existing model routines directly as agents

A solution: the Observer Pattern



Subscriber variants

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

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

click.subscribe (**agent** *your_procedure* (*a*, ?, ?, *b*))

click.subscribe (**agent** *other_object.other_procedure*)

A word about tuples

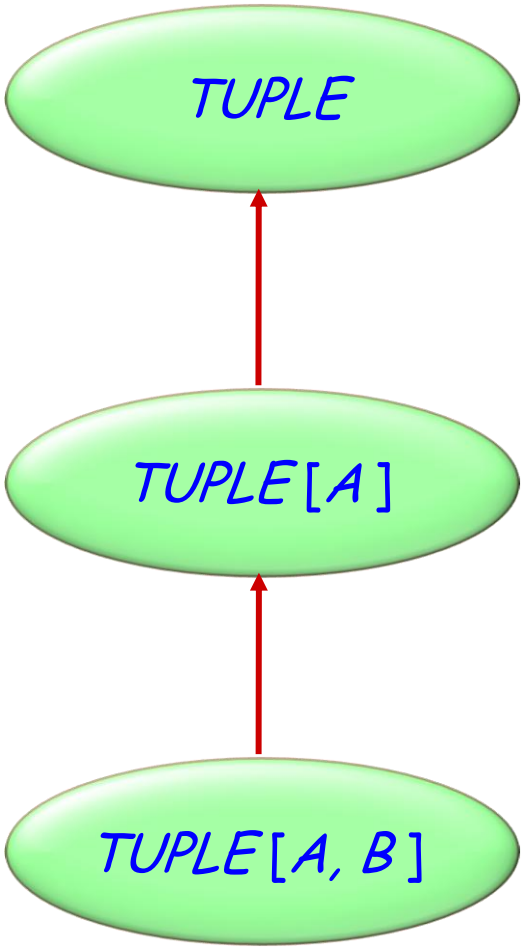
Tuple types (for any types *A*, *B*, *C*, ...):

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*]

Tuple type inheritance



Labeled tuple types

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

Example tuple:
[" *Tolstoi* ", *1865*, " *War and Peace*"]

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

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

but facilitates access to individual elements:

- To access tuple elements: e.g. *t.year*
- To modify tuple elements: *t.year* := *1866*

Labeled tuples amount to a restricted form of (anonymous) class. Exercise: write the class equivalent for the above.

What you can do with an agent *a*

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

A manifest tuple

```
a.call ([horizontal_position, vertical_position])
```

If *a* is associated with a function, *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.call* ([*argument_tuple*])

  calls the procedure

- If *a* represents a function, *a.item* ([*argument_tuple*])

  calls the function and returns its result

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 manifest tuple

```
a.call ([horizontal_position, vertical_position])
```

If *a* is associated with a function, *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:

```
u := agent a0.f (a1, a2, a3) -- All closed (as before)
w := agent a0.f (a1, a2, ?)
x := agent a0.f (a1, ?, a3)
y := agent a0.f (a1, ?, ?)
z := agent a0.f (?, ?, ?)
```

Calling the agent

```
f (x1: T1; x2: T2; x3: T3)
a0: C; a1: T1; a2: T2; a3: T3
```

`u := agent a0.f (a1, a2, a3)`	`u.call ([])`
`v := agent a0.f (a1, a2, ?)`	`v.call ([a3])`
`w := agent a0.f (a1, ? , a3)`	`w.call ([a2])`
`x := agent a0.f (a1, ?, ?)`	`x.call ([a2, a3])`
`y := agent a0.f (?, ?, ?)`	`y.call ([a1, a2, a3])`

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(?, u, v), a, b)
```

The integration function

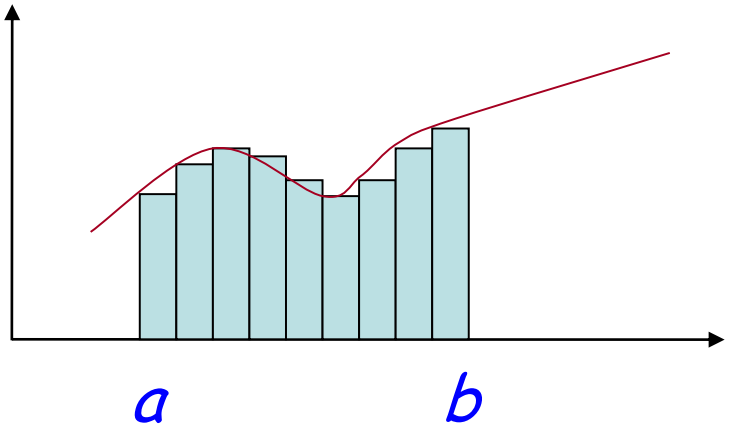
```
integral (f : FUNCTION [ANY, TUPLE [REAL], REAL];
          a, b : REAL): REAL
        -- Integral of f over  interval [a, b].
    local
        x: REAL; i: INTEGER
    do

        from x := a until x > b loop

                Result := Result + f.item ([x]) * step
                i := i + 1
                x := a + i * 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
```

```
class EMPLOYEE feature
        is_married: BOOLEAN
        ...
end
```

Iterators

In class *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)

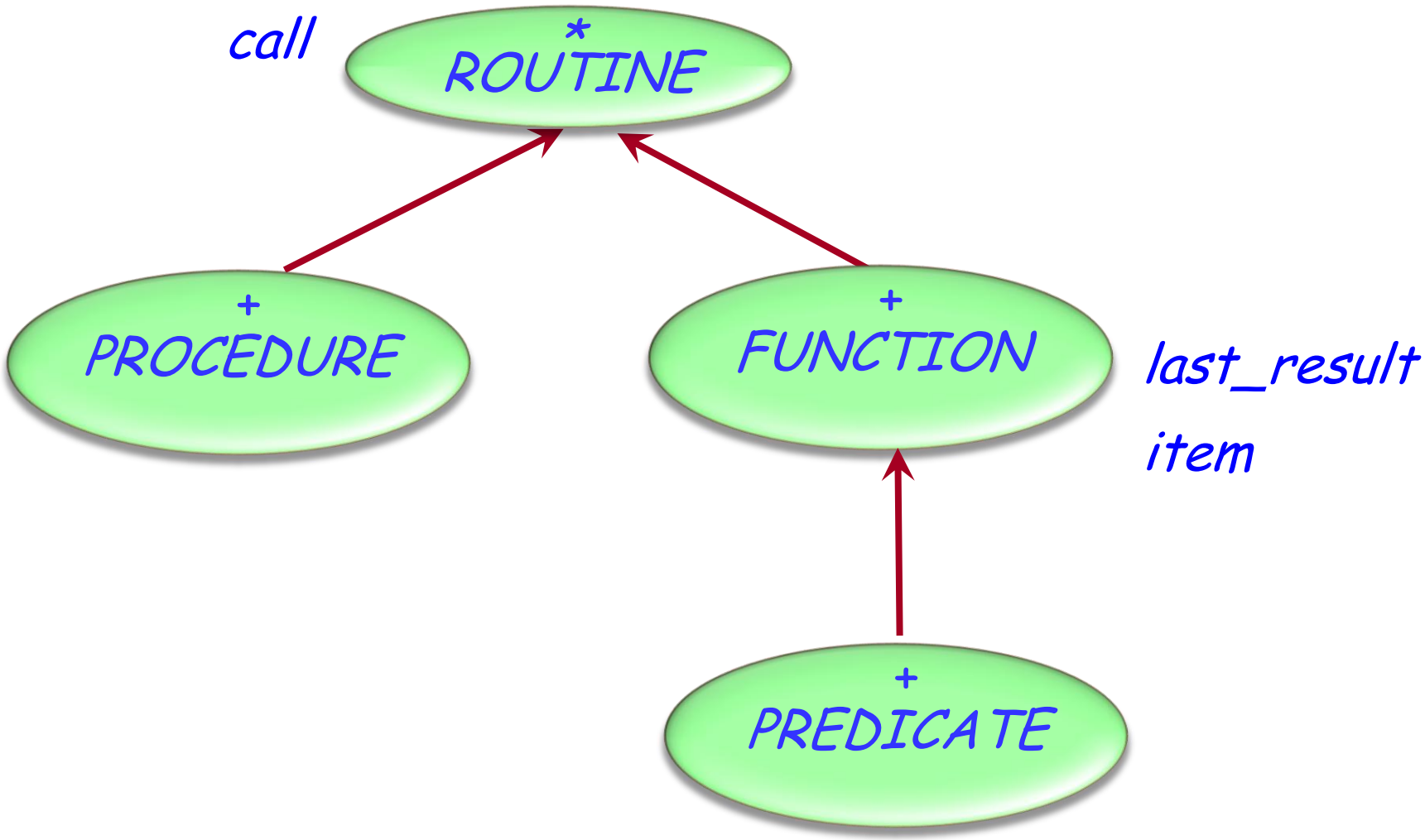
Iteration

High-level contracts

Numerical programming

Introspection (finding out properties of the program itself)

Kernel library classes representing agents



Declaring an agent

```
p: PROCEDURE [ANY, TUPLE]
    -- Agent representing a procedure,
    -- No open arguments.

q: PROCEDURE [ANY, TUPLE [X, Y, Z]]
    -- Agent representing a procedure,
    -- 3 open arguments.

f: FUNCTION [ANY, TUPLE [X, Y, Z], RES]
    -- Agent representing a function,
    -- 3 open arguments, result of type RES.
```

Calling the agent

```
f (x1: T1; x2: T2; x3: T3)
a0: C; a1: T1; a2: T2; a3: T3
```

`u := agent a0.f (a1, a2, a3)`	`u.call ([])`
`v := agent a0.f (a1, a2, ?)`	`v.call ([a3])`
`w := agent a0.f (a1, ? , a3)`	`w.call ([a2])`
`x := agent a0.f (a1, ?, ?)`	`x.call ([a2, a3])`
`y := agent a0.f (?, ?, ?)`	`y.call ([a1, a2, a3])`

Type of an agent

```
f (x1: T1; x2: T2; x3: T3)
a0: C; a1: T1; a2: T2; a3: T3
```

u := agent a0.f (a1, a2, a3)

v := agent a0.f (a1, a2, ?)

w := agent a0.f (a1, ? , a3)

x := agent a0.f (a1, ?, ?)

y := agent a0.f (?, ?, ?)

What we have seen

The event-driven mode of programming, also known as publish-subscribe

The Observer pattern

Agents (closures, delegates…): encapsulating pure behavior in objects

Applications to numerical programming and iteration