To be cited as follows: Bertrand Meyer, *The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design*, to appear in *Festschrift in Honor of Ole-Johan Dahl*, eds. Olaf Owe et al., Springer-Verlag, Lecture Notes in Computer Science 2635, 2003.

# The power of abstraction, reuse and simplicity: An object-oriented library for event-driven design

## ABSTRACT

A new library for event-driven design, defining a general and extendible scheme yet easy to learn and use on both the publisher and subscriber sides, provides an opportunity to analyze such other approaches as the "Observer Pattern", the event-delegate mechanism of .NET and its "Web Forms", then to draw some general software engineering lessons.

## 1 OVERVIEW

**Event-driven** software design avoids any direct connection, in a system's architecture, between the unit in charge of executing an operation and those in charge of deciding when to execute it.

Event-driven techniques have gained growing usage because of their flexibility. They are particularly common for Graphical User Interface applications, where the operations come from an application layer and the decision to execute them comes from a GUI layer in response to events caused by human users. An event-driven scheme can shield the design of the application layer from concerns related to the user interface. Many application areas other than GUI design have used these ideas.

Closely related techniques have been proposed under such names as *Publish-Subscribe* and *Observer Pattern*.

This article describes the Event Library, a reusable component solution of broad applicability, covering all these variants. Intended to be easy to learn, the library consists in its basic form of one class with two features, one for the production of events and one for their consumption.

The discussion will compare this solution to the Observer Pattern and mechanisms recently introduced by .NET. It will expand on this analysis to examine more general issues of software engineering, including the role of abstraction, the transition from design patterns to reusable components, the concern for simplicity, and the contribution of object technology.

Section 2 quickly presents the essentials of the Event Library. Section 3 explains event-driven design and what makes it attractive. Sections 4, 5 and 6 analyze other solutions: the Observer Pattern, the .NET event handling mechanism, the Web Forms library of ASP.NET. Section 7 gives the remaining details of the Event Library. Section 8 examines the software engineering issues that led to this work and draws general conclusions.

# 2 EVENT LIBRARY ESSENTIALS

The Event Library consists at its core of one class, *EVENT_TYPE* with a feature *publish* for publishers and a feature *subscribe* for subscribers. The library is written in Eiffel; so are the usage examples in this section.

First the subscriber side. Assume you want to ensure that any future mouse click will cause execution of a certain procedure of your application

```
your_procedure (a, b: INTEGER)
```

passing to it the mouse coordinates as values for *a* and *b*. To obtain this effect, simply *subscribe* the desired procedure to the event type *mouse_click*:

```
mouse_click.subscribe (agent your_procedure)                    /1/
```

The argument to *subscribe* is an "agent" expression; an agent in Eiffel is an object representing a routine, here *your_procedure*.

In most cases this is all one needs to know on the subscriber side, for example to produce a graphical application using a GUI library. An advantage of the scheme is that it lets you start from an existing system and add an event-driven scheme such as a GUI without writing any connecting code. You'll just reuse the existing routines directly, linking them to event types through agent expressions as above. This extends to routines with extra arguments: assuming

```
other_procedure ( text: STRING; a, b: INTEGER; date: DATE )
```

you can still subscribe the procedure without any "glue code" through

```
mouse_click.subscribe (agent other_procedure ("TEXT", ?, ?, Today))    /2/
```

where the question marks indicate the values to be filled in by the event. (The agent in form **/1/** can be written more explicitly as **agent** *your_procedure* (**?**, **?**).)

So much for subscribers. The basic scheme for publishers is also straightforward. To trigger a mouse click event, all the GUI library will do is

```
mouse_click.publish ([x_position, y_position])                  /3/
```

It is also the publisher's responsibility to declare *mouse_click* and create the corresponding object. It can take care of both through

```
mouse_click: EVENT_TYPE [TUPLE [INTEGER, INTEGER]] is
        once
                create Result
        end                                                     /4/
```

Class *EVENT_TYPE* is generic; the parameter *TUPLE* [*INTEGER*, *INTEGER*] indicates that a mouse click produces event data consisting of two integers, representing the mouse coordinates, collected into a two-element "tuple".

Since *mouse_click* just represents an ordinary object — an instance of class *EVENT_TYPE* — the instruction that creates it could appear anywhere. One possibility, as shown, is to put it in a "once function" defining *mouse_click*. A once function is executed the first time it's called, whenever that is, the same result being returned by every subsequent call. This language mechanism addresses the issue of providing initialization operations without breaking the decentralized architecture of well-designed O-O systems. Here it creates the mouse click object when first needed, and retains it for the rest of the execution.

The scheme as described covers global events: the subscriber call **/1/** subscribes *your_procedure* to any mouse click anywhere. Instead we may want to let subscribers select events in a given graphical element such as a button. We simply turn *mouse_click* into a feature of class *BUTTON*, so that subscriber calls will be

```
your_button.mouse_click.subscribe (agent your_procedure)          /5/
```

perhaps clearer as *your_button_click*.*subscribe* (**agent** *your_procedure*), retaining the original form **/1/** with *your_button_click* set to *your_button*.*mouse_click*.

What we have just seen defines, for the majority of applications, the user's manual of the Event Library:

- On the publisher side, declare and create the event type object; trigger a corresponding event, when desired, by calling *publish*.
- On the subscriber side, call *subscribe* with an agent for the desired routine.

Only one class is involved, *EVENT_TYPE*; there is no need to define specific classes for each event type (mouse click, mouse movement etc.) as, for example, in the .NET model studied below — although you can do so if you wish by introducing descendants of *EVENT_TYPE* that specialize the event data. There is also no need for the publishers or the subscribers to inherit from any particular classes, such as the abstract classes *SUBJECT* and *OBSERVER* of the Observer Pattern, also studied below.

Section 7 will describe some of the more specialized features of the library. As is often the case when the basic design of a library uses a small number of abstractions tailored to the problem, it is possible to add special-purpose facilities without disturbing users who need only the basics.

To understand the rationale behind this design, we will now step back to examine the general issues of event-driven computation, and some previously proposed solutions.

# 3 EVENT-DRIVEN DESIGN

Event-driven design offers interesting architectural solutions when execution must respond to events whose order is hard to predict in the program text.

## Putting the user in control

GUI and WUI (Web User Interfaces) provide the most immediately visible illustration of why an event-driven scheme may be useful.

Consider this piece of WUI built with ASP.NET (the Web programming library for .NET):


The interface that we show to our user includes a text field and a button. There might be many more such "*controls*" (the Windows term for graphical elements, called "widgets" in the X Window system). We expect that the user will perform some input action, and we want to process it appropriately in our program. The action might be typing characters into the text field, clicking the button, or any other, such as menu selection, using controls not shown above.

But which of these will happen first? Indeed, will any happen at all?

We don't know.

In the early days, the problem didn't exist. Programs would just read user input, using for example a loop to consume successive lines, as in

**[A]**

```
from
        read_line
        count := 0
until
        last_line.empty
loop
                count := count + 1
                        -- Store last_line at position count in Result:
                Result.put (last_line, count)
                read_line
end
```

This was good enough when we had a single sequential input medium and the program was in charge of deciding when, where and how to enjoin the user to enter some input, for example on a teletype console.

With current input techniques, a user sitting at a graphics workstation is free at any time to go to the text field, the button or any other control. He, not the program, is in control.

To support such modes of interaction, event-driven programming replaces a *control structure* by a *data structure*. A control structure means that the program decides when to execute things. Instead we want to let the user decide what to do next. We'll call these user decisions *events*. The system will use a data structure — let's call it the **event-action table** — to record in advance the actions that it has to execute when events of specific types occur. After that it relies on the event handling machinery to watch for events of recognized types and, when detecting one of them, trigger the corresponding action as found in the event-action table.

> A role remains for control structures: each operation, while it executes, defines the scheduling of its own operations, using a control structure that can be arbitrarily complex. But when the operation terminates the event-driven scheme takes over again.

Overall, it's a major change. The program has relinquished direct control of global execution scheduling to a generic table-driven mechanism. For best results that mechanism should be a library, for example a GUI or WUI library, or — more generic yet — the Event Library, not tied to any specific application area.

This yields a clear division of tasks between such a general-purpose library and any particular application. The application is in charge of recording event-action associations; when the true show begins, the library is in charge of catching and processing events.

Application authors have their say, since what gets executed in the end are the actions — taken from the program — that they have planted in the table. But they do not directly control the scheduling of steps.

The library owns the event-action table, so that application programmers should not need to know anything about its implementation. With the Event Library they simply record event-action associations, through calls to *subscribe*; the library takes care of maintaining these associations in the appropriate data structure. We'll see that in some other frameworks, such as .NET, programmers work at a lower level of abstraction, closer to the internal representation of the event-action table.

## Publishers and subscribers

The overall scheme of programming in an event-driven style is this:

1 • Some part of the system is able to trigger events. We call it a **publisher**.

2 • Some part of the system wants to react to these events. We call it a **subscriber**. (“Observer” would also do, as in the “Observer Pattern”, where the publisher is called a “subject”.)

3 • The subscriber specifies actions that it wants to execute in connection with events of specified types. We’ll say that the subscriber **registers** an action for an event type. The effect of registration is to record an association between an event type and a subscriber into the event-action table. Registrations usually happen during initialization, but subscribers can continue to register, or de-register, at any time of the execution; that’s one of the advantages of using a table-driven scheme, since the table can be modified at any time.

4 • At any time during execution proper, after initialization, the publisher can trigger an event. This will cause execution of the routines that any registered subscribers have associated with the event’s type.

For this discussion we must be careful about distinguishing between *events* and *event types*. The notion of mouse click is an event type; a user clicking his mouse will cause an event. Although the data structure is called the *event*-action table for brevity, its definition clearly specified that it records information about event types. Publishers, on the other hand, trigger events, each of a certain type.

Healthy skepticism should lead us to ask why we need all this. Instead of an indirect relationship through an event-action table, couldn’t we just skip step 3 and let, in step 4, the subscriber call the publisher, or conversely?

A subscriber can indeed call its publishers directly through a generalization of the earlier sequential reading scheme: it will listen to events of several possible types rather than just one, pick up the first one that happens, select the appropriate action, and repeat. This has, however, two limitations. One is that you need to put the subscriber in charge of the application’s control structure; that is not always appropriate. Another, more serious, is that it is not easy with this scheme to ensure that events raised by a publisher trigger actions in *several* subscribers.

← *[A], page 4.*

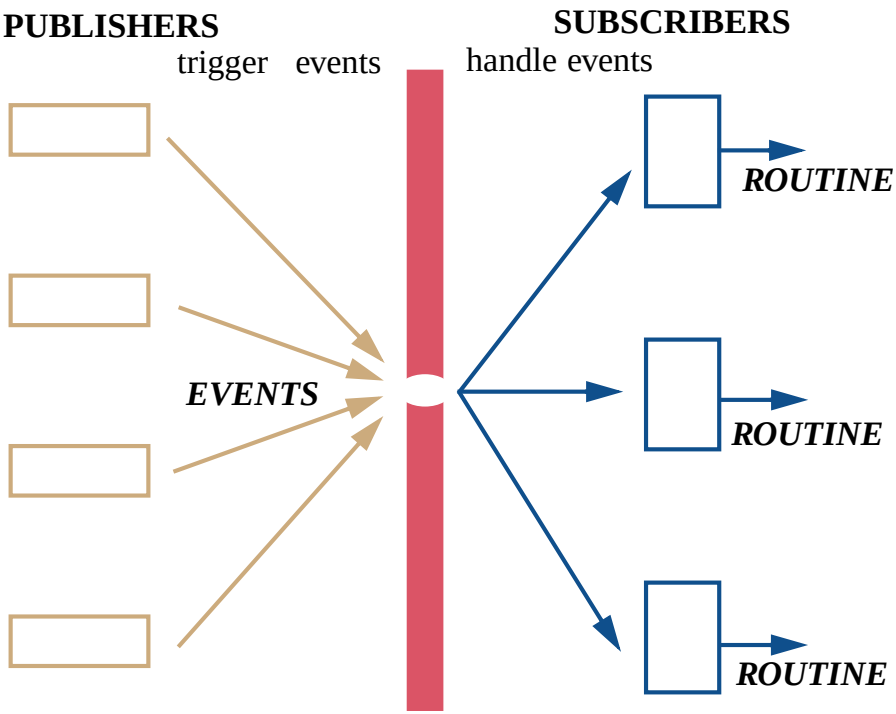
Alternatively, the publisher could call the subscriber's routine directly

```
my_subscriber.routine (my_arguments)
```

using the standard object-oriented call mechanism. This works as long as the whole scheme is static: the publishers know their subscribers, and this information is defined once and for all so that publishers' code can include calls such as the above for each subscriber to each event type.

The limitations of both solutions indicate where event-driven programming becomes interesting. We may picture the general situation as one of those quantum physics experiments that bombard, with electrons or other projectiles, some screen with a little hole:

*Triggering and handling events*




The event-driven scheme decouples the subscribers from the publishers and may be attractive if one or more of the following conditions hold:

- *Publishers can't be required to know who the subscribers are*: they trigger events, but do not know who is going to process those events. This is typically the case if the publisher is a GUI or WUI library: the routines of the library know how to detect a user event such as a mouse click, but they should not have to know about any particular application that reacts to these events, or how it reacts. To an application, a button click may signal a request to start a compilation, run the payroll, or shut down the factory. To the GUI library, a click is just a click.

- *Subscribers may register and deregister while the application is running*: this generalizes the previous case by making the set of subscribers not only unknown to publishers but also variable during execution.

- *Any event triggered by one publisher may be consumed by several subscribers*. For example the event is the change of a certain value, say a temperature in a factory control system; then the change must be reflected in many different places that "observe" it, for example an alphanumeric display, a graphical display, and a database that records all historical values. Without an event mechanism the publisher would have to call routines in every one of these subscribers, causing too much coupling between different parts of the system. This would mean, in fact, that the publisher must know about all its subscribers, so this case also implies the first one.

- *The subscribers shouldn't need to know about the publishers*: this is less commonly required, but leads to the same conclusions.

In all such cases the event-driven style allows you to build a more flexible architecture by keeping publishers and subscribers at bay.

There is a downside: if you are trying to follow the exact sequence of run-time operations — for example when debugging an application — you may find the task harder, precisely because of the indirection. A plain call $x.f\,(\ldots)$ tells you exactly what happens: after the preceding instruction, control transfers to $f$, until $f$'s execution terminates, and then comes back to the instruction following the call. With an instruction that triggers an event, all you know is that some subscribers may have registered some routines to handle events of that kind, and if so they will execute these routines. But you don't immediately know who they are; indeed they may vary from execution to execution. So it is more delicate to track what's going on. One should weigh this objection — which some authors have proposed to address by replacing event-driven design with techniques inspired by parallel programming **[18]** — before deciding to embark on an event-driven architecture.

## Controls

In cases such as GUI and WUI programming, the event-action table will generally contain not mere pairs — actions coupled with event types — but **triples**: we don't just specify "for events of this type, execute that action", but "for events of this type occurring in this *control*, execute that action", as in:

- "*If the user clicks the EXIT button, exit the application*".
- "*If the mouse enters this window, change the border color to red*".
- "*If this sensor reports a temperature above 40*$^{o}$ *C, ring the alarm*".

In the first case the control is a button and the event type is "mouse click"; in the second, they are a window and "mouse enter"; in the third, a temperature sensor and a measurement report.

A "control" is usually just a user interface element. As the last example indicates, the concept also applies outside of the UI world.

A common library interface to let subscribers deposit triples into the event-action table (we'll continue to call it that way) uses calls of the conceptual form

*record_association* (*some_control*, *some_event_type*, *some_action*) **/6/**

and leaves the rest to the underlying GUI or WUI machinery. That's the essence of event-driven programming as supported by many modern graphics toolkits, from Smalltalk to EiffelVision to the Windows graphical API and the Web Forms of .NET. The most common variant is actually

*add_association* (*some_control*, *some_event_type*, *some_action*)

which adds *some_action* to the actions associated with *some_event_type* and *some_control*, so that you can specify executing *several* actions for a given event-control pair. We'll retain the first form **/6/** since it corresponds to the most frequent need; it includes the second one as a special case if we assume a provision for composite actions.

The Event Library seemed at first not to support controls since the basic mechanism *mouse_click*.*subscribe* (...) **/1/** did not make them explicit; but we saw that it's just a matter of making an event type belong to a control object, then use *your_button*.*mouse_click*.*subscribe* (...) **/5/**, which directly provides the general scheme **/6/**.

## Actions as objects

In a classical O-O language, we have a problem. Even though we don't need to manipulate the event-action table directly, we know it will exist somewhere, managed by a graphical library, and that it's a data structure — a structure made of objects, or (more realistically) references to objects. In each entry we expect to find a triple containing references to:

- One control.
- One event type.
- One action — or, as a result of the last observation, one list of actions.

Are these things objects? Controls, definitely. Any graphical O-O library provides classes such as *WINDOW* and *BUTTON* whose instances are objects representing controls — windows, buttons and so on. Event types too can be defined as objects in an O-O language; we saw how the Event Library does it. But what about actions?

Actions are given by our program's code. In an O-O program, the natural unit for an action is a routine of a class. But a routine is not an object.

This won't be too much of a concern for a C++ programmer, who may just use a *function pointer*: an integer denoting the address where a routine's code is stored, providing a way to execute the routine. But that's not type-safe, since one can do too many other things with the pointer. As a consequence, O-O languages intent on providing the benefits of static typing do not provide function pointers.

The notion of *agent* used in the Event Library is an object-oriented mechanism that addresses the issue within the constraints of type checking. An Eiffel agent is an object that represents a routine ready to be called.

> Some of its operands (target and arguments) can be fixed, or *closed*, at the time the agent is defined; the others, called *open* operands and expressed — when needed — as question marks **?** in our earlier examples, must be provided at the time of each call. In **agent** *some_routine* all arguments are open; in **agent** *some_routine* (*1*, **?**, **?**, "*SOME TEXT*") the first and last arguments are closed, the others open. You can also make the *target* open, as in **agent** {*TARGET_TYPE*}**.***some_routine* (*1*, **?**, **?**, "*SOME TEXT*")

Some languages provide comparable mechanisms under the name "block" or "closure". The "delegates" of .NET and C# are a limited form of agent where arguments are always open and the target is always closed.

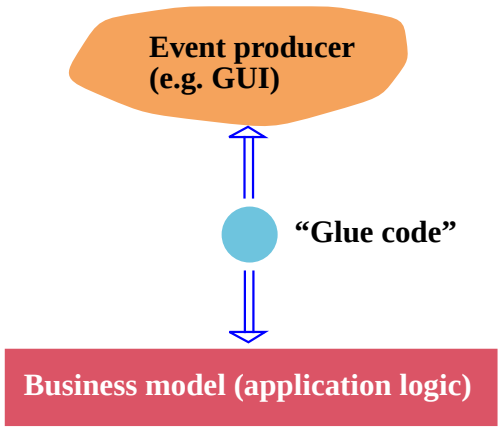
Java doesn't have such notion, meaning that to represent an action as object you have to create a little class that includes the corresponding routine. The availability of "nested classes" limits the amount of code that must be written for such classes, but the solution lacks extendibility and scalability.

## Avoiding glue

When building an event-driven application, you will need at some stage to connect the subscribers with the publishers. One of the guiding concerns — reflected in the design of the Event Library — must be to keep such connections as light as possible.

This goal is particularly relevant to the common case of restructuring an existing application to give it an event-driven architecture. The application may provide many functions, perhaps developed over a long period and embodying a sophisticated "business model" for a certain domain. The purpose of going event-driven might be to make these functions available through a graphical or Web interface, taking advantage of an event-driven GUI or WUI library. In this case both the business model and the library predate the new architecture.

***Connecting publishers and subscribers***




Common names for the three parts appearing on the figure follow from the Smalltalk "MVC" scheme that inspired many event-driven GUI designs: *Model* for the existing application logic, *View* for the user interface, and *Controller* for the connection between the two.

With such terminology the above goal is easily stated: we seek to get rid of the Controller part, or reduce it to the conceptually inevitable minimum.

The Event Library offers two complementary styles to achieve this. In the first style, we let the application consume events by becoming a subscriber through calls of the form seen earlier

```
some_event_type.subscribe (agent some_routine)                    /7/
```

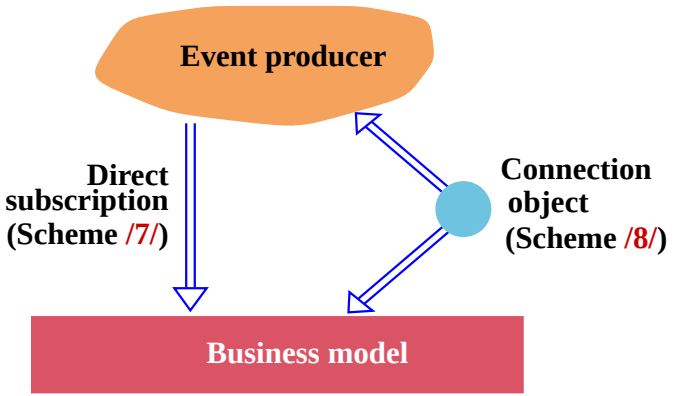
explicitly making the consumer application event-driven.

In many cases this is appropriate. But what if you want to reuse *both* the event producer and the event consumer (the application) exactly as they are? The Event Library and the agent mechanism allow this too. You'll leave both the producer and the consuming application alone, connecting application routines to producer events through a simple intermediary. Just add an explicit target to the agent expression: instead of **agent** *some_routine* as used above, which denotes the routine ready to be applied to the current object, you may select any other object as target of the future call:

```
some_event_type.subscribe (agent other_object.some_routine)          /8/
```

By using either form, you can select the style that you prefer:

***Two connection styles***




Either may be appropriate depending on the context:

- Scheme **/7/** adds event-driven scheduling to an existing application by plugging in routines of that application directly.
- Scheme **/8/** lets you reuse both pre-existing event producers (publishers) and pre-existing event consumers (subscribers), unchanged. By definition, you'll need some glue code then. Scheme **/8/** reduces it to the minimum possible: calls to *subscribe* using agents with explicit targets. One of the benefits of this style is that it lets you provide *several* interfaces to a given application, even within a single program.

To reduce the need for glue code even further, you may take advantage of the agent mechanism's support for combining *closed* and *open* arguments. Assume that an existing meteorological application includes a routine

```
show_rainfall (x, y: INTEGER; start, end : DATE): REAL
        -- Display amount of rain recorded between start and end at coords x, y.
```

You build a graphical interface that shows a map with many cities, and want to ensure that once a user has chosen a starting and ending dates *Initial* and *Final* moving the mouse across the map will at each step display the rainfall at the corresponding map position. This means that when calling the procedure *show_rainfall* for each *mouse_move* event we should treat its first two arguments differently from the other two:

- The application sets *start* and *end* from *Initial* and *Final*.
- The GUI library mechanism will fill in a fresh *x* and *y* — event data — each time it triggers the event.

To achieve this effect, simply subscribe an agent that uses open arguments for the first set and closed arguments for the second set, as in **/2/**:

```
mouse_move.subscribe (agent show_rainfall (?, ?, Initial, Final))          /9/
```

(This could also use an explicit target as in **/8/**; the target could be closed or itself open.) The generality of this mechanism lets you tweak an existing routine to fit a new context: the subscriber freezes certain operands at subscription time, and leaves the others for the publisher to provide, as event data, at the time of event publication.

The benefit here is that the agent lets us reuse an existing four-argument routine, *show_rainfall*, at a place were we need a routine with only two arguments.

With other mechanisms such as the ones studied later in this chapter we would have to use two variables and write an explicit wrapper routine:

```
Initial, Final: DATE
        -- Start and end of rainfall data collection period

show_rainfall_at_initial_and_final (x, y: INTEGER) is                       /10/
        -- Display amount of rain recorded at x, y between Initial and Final.
    do
            show_rainfall (Initial, Final, x, y)
    end
```

For a few routines and event types this approach is acceptable. When scaled up to real applications, it generates noise code that pollutes the architecture, making the program harder to understand, maintain and extend.

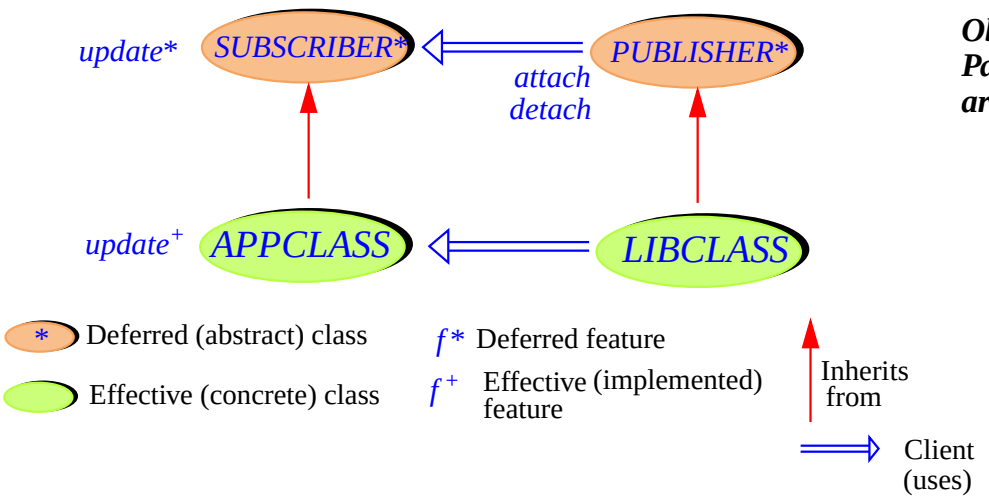
Agents and the Event Library help avoid these pitfalls and build stable solutions on both the publisher and subscriber sides, with minimum connection code between the two.

# 4 THE OBSERVER PATTERN

To provide more perspective on event-driven architectures and the design of the Event Library, this section and the next two examine other approaches.

First, the "Observer Pattern". As presented in the book *Design Patterns* **[6]**, it was one of the first descriptions of a general event-driven scheme.

The following figure illustrates the general structure of that solution. For ease of comparison with the rest of this article the names Observer and Subject used in the original have been changed to *SUBSCRIBER* and *PUBLISHER*. *APPCLASS* and *LIBCLASS* denote typical examples of effective (concrete) classes, one describing a typical subscriber and the other a typical publisher.




***Observer Pattern architecture***

Surprisingly, in the basic design it's the publishers that know about their observers, as shown by the direction of the client links on the figure: a publisher gets a new observer through the procedure *attach* and can remove it through *detach*. There is no equivalent to *register* on the subscriber side; in the primary example illustrating the pattern **[6]** — a clock that publishes ticks and two subscribers, both of them clock displays, one analog and the other digital — the subscriber objects get created with respect to a publisher, through a constructor (creation procedure) of the form

```
make (p: PUBLISHER) is
        -- Initialize this object as a subscriber to subject.
    do
        p.attach (Current)
    end
```

*Current is the current object ( also called self, this, Me in various O-O languages).*

so that the digital clock display gets created as

```
create digital_display.make (clock)
```

(in C++/Java/C# style: digitalDisplay = new DigitalDisplay (clock)).

An immediately visible limitation of the pattern is that it lacks a general solution for the publisher to pass information to the subscriber. That doesn't matter if all that characterizes an event is that it occurs. But many events will also need, as we have seen, to transmit *event data*, such as the mouse position for a mouse click. The Patterns book notes this issue ((**[6]**, page 298) and mentions two models for passing information, "push" and "pull", each implying even more coupling between the publisher and the subscriber. Each requires extra coding on both sides, taking into account the specific type of information being passed.

No reusable solution seems possible here — short of an explosion of the number of classes, as will be seen in the .NET model's approach in the next section — without both *genericity* and a *tuple type* mechanism as used by the Event Library. The Event Library represents event data through the generic parameter to *EVENT_TYPE*: when you introduce a new event type that generates, say, three pieces of information of types *A*, *B* and *C*, you will declare it of type *EVENT_TYPE* [*TUPLE* [*A, B, C*]]. Then the routine that you subscribe must take arguments of types *A, B, C*. This takes care of the connection, but is not possible in the C++/Java framework, used in most published discussions of patterns with the exception of the work of Jézéquel, Train and Mingins **[8]**.

The Observer Pattern design raises two other immediate objections:

- The creation procedure — *make* or the corresponding C++ constructor — must be written anew for each subscriber class, leading to useless repetition of code, the reverse of object technology principles.
- It's too restrictive to force subscribers to register themselves with publishers at creation time, not only because subscriber classes may already exist and have other creation requirements, but also because the subscribers should be able to subscribe later.

We can alleviate both criticisms, at least in part, by adding to the class *SUBSCRIBER* a procedure *subscribe* which, on the subscriber side, mirrors what *attach* provides on the publisher side, and uses *attach* in its implementation:

```
subscribe (p: PUBLISHER) is
        -- Attach current subscriber to p.
    do
        p.attach (Current)
    end
```

This extension explicitly makes subscribers clients of their publishers. But other problems remain.

One is that a publisher sends notification of an event by calling a procedure *update* in each subscriber. The procedure is declared as deferred (abstract) in class *SUBSCRIBER* and effected (implemented) in each specific observer class such as *APPCLASS* to describe the subscriber's specific reaction to the event. Each publisher keeps a list of its subscribers, updated by *attach* and *detach*; when it triggers an event, it traverses this list and calls *update* on each element in turn, relying on dynamic binding to ensure that each subscriber uses its own version. But this means that altogether a subscriber may register only one action! As a consequence, it may subscribe to only one type of event, except for the trivial case of handling several event types in the same way. This is severely restrictive. An application component should be able to register various operations to various publishers. With the Event Library design there is no limit, for a subscriber, to the number of calls of the form *some_control*.*some_event_type*.*subscribe* (**agent** *some_routine*).

The discussion in the Patterns book acknowledges the issue and proposes a solution (**[6]**, page 297): add one argument to *update* to represent the publisher, which will pass ***Current*** when calling *update* to let the subscriber discriminate between publishers. But this is not satisfactory. Since there is still just one *update* procedure in each subscriber, that procedure will have to know about all relevant publishers and discriminate between them — a step back to the kind of know-them-all, consider-each-case-in-turn decision schemes from which object technology tries to free software architectures. Worse, this means a new form for procedure *update*, with one extra argument, invalidating the preceding class design for *PUBLISHER* and implying that we can't have a single reusable class for this concept. Reusability concerns yield to general guidelines that have to be programmed again, in subtly different ways, for each new application.

More generally the need for the subscribers to know the publishers is detrimental to the flexibility of the architecture. The direction of this knowledge relation is not completely obvious, since the last figure, drawn before we added *subscribe* to the pattern, only showed (like the corresponding UML diagram in the original Observer Pattern presentation) a client link from the publisher to the subscriber. That link indicates that each publisher *object* keeps a list of its subscribers. It has no harmful effect on the architecture since the text of publisher *classes* will, properly, not list subscribers.

Subscriber classes, however, do mention the publisher explicitly. In the pattern's original version, that's because the constructor of a subscriber uses the publisher as argument; our addition of *subscribe* as an explicit procedure made this requirement clearer. It causes undesirable coupling between subscribers and publishers. Subscribers shouldn't need to know which part of an application or library triggers certain events.

Yet another consequence is that the Observer Pattern's design doesn't cover the case of a single event type that any number of publishers may trigger. You subscribe to *one* event type from *one* publisher, which the subscriber's text must name explicitly.

It is also not clear with the Observer Pattern how we could — as discussed — connect without glue code an existing business model and an existing event-driven interface library.

← *"Avoiding glue", page 11*..

The Event Library overcomes all these limitations: publishers publish events of a certain event type; subscribers subscribe to such event types, not to specific publishers. The two sides are decoupled. All the specific schemes discussed in the Observer Pattern presentation are still possible as special cases. For all this extra generality, the interface is considerably simpler; it involves no abstract class and no inheritance relationship; it places no particular requirement on either subscriber or publisher classes, which can be arbitrary application or library elements. All a class must do to become a publisher or subscriber is to create objects of the appropriate type and call the desired features on them. In addition the solution is based on a reusable class, not on a design pattern, meaning that it does not require programmers to code a certain scheme from a textbook and change some aspects (such as the arguments to *update*) for each new application; instead they just rely on ready-to-use components.

It is legitimate to ask what caused the design of the Observer Pattern to miss the solution described here in favor of one that appears to be harder to learn, harder to use, less powerful, less reusable, and less general. Some of the reasons seem clear:

- Although solutions had been published before, the Design Patterns book was one of the first times the problem was approached in its generality, so it's not surprising that it didn't produce the last word. The simplest solution doesn't always come first, and it is easier to improve an original idea than to come up with it in the first place. This observation is also an opportunity to note that this article's technical criticism of the Observer Pattern and other existing designs only make sense when accompanied by the obvious acknowledgment of the pioneering insights of the patterns work and other more recent developments.

- The simpler and more advanced solution is only possible, as we have already noted in the case of event data representation, because of advanced language features: genericity, tuples, agents. The work on design patterns has been constrained by its close connection with C++ and then Java, both of which lack some of these features while sometimes adding their own obstacles. Take this comment in the presentation of the Observer Pattern: "*Deleting a* [publisher object] *should not produce dangling references in its* [subscribers]" (**[6]**, page 297), followed by suggestions on how to avoid this pitfall. This reflects a problem of C++ (lack of standard garbage collection support) and has an adverse effect on the pattern.

- The Patterns work endeavors to teach programmers useful schemes. This is far from the goal of object technology, the Eiffel method in particular, which seeks to build reusable components so that programmers do *not* have to repeat a pattern that has been identified as useful.

We may also see another reason as possibly even more fundamental. The Observer Pattern design *uses the wrong abstractions* and in the process misses the right abstraction. Talk of right and wrong may sound arrogant, but seems justified here in light of the results. The abstractions "Subscriber" and "Publisher" (Subject and Observer in the original), although attractive at first, turn out to be too low-level, and force application designers to make relevant classes inherit from either *SUBSCRIBER* or *PUBLISHER*, hampering the reuse of existing software elements in new subscribing or publishing roles. Choosing instead *Event type* as the key abstraction — the only one introduced so far for the Event Library — leads to a completely different design.

With all the attraction of new development tools, concepts and buzzwords, it is easy to forget that the key to good software, at least in an object-oriented style (but is there really any other path?), is a task that requires insight and sweat: the discovery of good abstractions. The best hope for dramatically decreasing the difficulty and cost of software development is to capture enough of these abstractions in reusable components. Design patterns are a useful and sometimes required first step of this effort, but are not sufficient since they still require each developer to learn the patterns and reimplement them. Once we have spotted a promising pattern, we shouldn't stop, but continue to refine the pattern until we are able to turn it into a library ready for off-the-shelf reuse.

## 5 THE .NET EVENT-DELEGATE MODEL

Probably inspired by the Observer Pattern but very different in its details, an interesting recent development for the spread of event-driven design is the Microsoft .NET object model, which natively supports two concepts directly related to event-driven design: *delegates*, a way to encapsulate routines into objects, and *event* types. The presence of these mechanisms is one of the principal differences between the .NET model and its most direct predecessor, the Java language; delegates, introduced in Microsoft's Visual J++ product, were at the center of Sun's criticism of that product **[19]**.

We will examine how events and delegates address the needs of event-driven design.

The technology under discussion is not a programming language but the ".NET object model" providing a common substrate for many different languages. (Another article **[14]** discusses how the common object model and the programming languages manage to support different languages with their own diverse object models.)

The .NET delegate-event model is quite complex, as we found out in surveying it for a book on .NET **[15]**. The present description is partial; for a complete picture see the book.

C# and Visual Basic .NET, the two main languages directly provided by Microsoft to program .NET, have different syntax generally covering similar semantics — the semantics of the .NET object model. Event-delegate programming is one of the areas where the two languages show some significant differences; we'll stay closer to the C# variant.

## The basis for delegates

The .NET documentation presents delegates as a type-safe alternative to function pointers. More generally, a delegate is an object defined from a routine; the principal operation applicable to such an object is one that calls the routine, with arguments provided for the occasion. That operation — corresponding to the feature *call* applicable to Eiffel agents — is called DynamicInvoke in the basic .NET object model.

The notion is easy to express in terms of the more general agent concept introduced earlier. Assuming, in a class *C*, a routine of the form

```
r (a1: A; b1: B)
```

with some argument signature, chosen here — just as an example — to include two argument types *A* and *B*, .NET delegates correspond to agents of the form

```
agent r                                                    /11/
```

or

```
agent x.r                                                  /12/
```

for *x* of type *C*. In either case, the delegate denotes a **wrapper object** for *r*: an object representing *r* ready to be called with the appropriate arguments. You would write such a call (still in Eiffel syntax for the moment) as

```
your_delegate.call ([some_A_value, some_B_value])          /13/
```

where *your_delegate* is either of the above agents. **/13/** has exactly the same effect as a direct call to *r*:

```
r (some_A_value, some_B_value)                             /14/
x.r (some_A_value, some_B_value)                           /15/
```

and hence is not interesting if it's next to the definition of *your_delegate*. The interest comes if the unit that defines *your_delegate* passes it to other modules, which will then execute **/13/** when *they* feel it's appropriate, without knowing the particular routine *r* that the delegate represents.

The uncoupling will often go further, in line with the earlier discussion: the defining unit inserts *your_delegate* into an event-action table; the executing unit retrieves it from there, usually not knowing who deposited it, and executes *call* — DynamicInvoke in .NET — on it.

At this stage things are very simple. A delegate is like a restricted form of agent with all arguments open. There is no way to specify some arguments as "closed" by setting them at agent definition time, and leave some others open for filling in at agent call time, as in our earlier example *mouse_move*.*subscribe* (**agent** *show_rainfall* (**?**, **?**, *Initial*, *Final*) **/9/**. To achieve that effect you would have to write a special wrapper routine that includes the open arguments only and fills in the closed arguments to call the original, in the style of *show_rainfall_at_initial_and_final* **/10/**.

The only difference between the delegate form without an explicit target, **/11/**, and the form with *x* as a target, **/12/**, is that the first is only valid in the class *C* that defines *r* and will use the current object as target of future calls, whereas the second, valid anywhere, uses *x* as target. The direct call equivalent is **/14/** in the first case and **/15/** in the second. There is no way with delegates to keep the target open, as in the Eiffel notation **agent** {*TARGET_TYPE*}.*r*; to achieve such an effect you have again to write a special wrapper routine, this time with an extra argument representing the target.

In the examples so far the underlying routine *r* was a procedure, but the same mechanisms apply to a delegate built from a function. Then calling the delegate will return a result, as would a direct call to *r*.

So the basic idea is easy to explain: a delegate is an object representing a routine ready to be called on a target set at the time of the delegate's definition and arguments set at the time of the call.

The practical setup is more complicated. A delegate in .NET is an instance of a class that must be a descendant of a library class, Delegate, or its heir MulticastDelegate, which introduces the feature DynamicInvoke. We won't go into the details — see **[15]** — because these classes and features are not for direct use. They have a special status in the .NET runtime; programmers may not write classes that inherit from Delegate. It's a kind of magic class reserved for use by compiler writers so that they can provide the corresponding mechanisms as language constructs. In Eiffel, the agent mechanism (not .NET-specific) readily plays that role. C# with the keyword **delegate** and Visual Basic .NET with **Delegate** provide constructs closely mapped to the semantics of the underlying .NET classes.

Here is how it works in C#. You can't directly define the equivalent of **agent** *r* but must first define the corresponding **delegate type**:

```
public void delegate AB_delegate (A x, B y);                    /16/
```

In spite of what the keyword **delegate** suggests, the declaration introduces a delegate type, not a delegate.

> The .NET documentation cheerfully mixes the two throughout, stating in places **[16]** that a delegate "*is a data structure that refers to a static method or to a class instance and an instance method of that class*", which defines a delegate as an object, and in others **[17]** that "*Delegates are a reference type that refers to a* Shared *method of a type or to an instance method of an object*" which, even understood as "*A* delegate *is*…", says that a delegate is a type. We'll stick to the simpler definition, that a delegate wraps a routine, and talk of **delegate type** for the second notion.

A delegate type declaration resembles a routine definition — listing a return type, here **void**, and argument types — with the extra keyword **delegate**.

Armed with this definition you can, from a routine r of a matching signature, define an actual delegate that wraps *r*:

```
AB_delegate r_delegate = new AB_delegate ( r );                 /17/
```

The argument we pass to the constructor is the name of a routine, r. This particular instruction is type-wise valid since the signatures declared for the delegate and the routine match. Otherwise it would be rejected. A delegate constructor as used here is the only context in which C#, and more generally the .NET model, allow the use of a routine as argument to a call.

Instead of **new** AB_delegate (r) you can choose a specific call target *x* by writing **new** AB_delegate ( x**.**r); or, if r is a static routine of class C, you can use **new** AB_delegate ( C**.**r). (A static routine in C++ and successors is an operation that doesn't take a target.)

All we have achieved so far is the equivalent of defining the agent expression **agent** *r* or **agent** *x***.***r*. Because of typing constraints this requires defining a specific delegate type and explicitly creating the corresponding objects. The noise is due to the desire to maintain type safety in a framework that doesn't support genericity: you must define a delegate type for every single routine signature used in event handling.

Equipped with a delegate, you'll want to call the associated routine (see **/13/**). This is achieved, in C#, through an instruction that has the same syntactic form as a routine call:

```
AB_delegate (some_A_value, some_B_value);                    /18/
```

Visual Basic .NET offers corresponding facilities. To define a delegate type as in **/16/** you will write:

```
Delegate Sub AB_delegate (x As A, y As B);
```

To create a delegate of that type and associate it with a routine as in **/17/**:

```
Dim r_delegate As New AB_delegate (AddressOf r);
```

The operator **AddressOf** is required since — unlike in C# — you can't directly use the routine name as argument.

## Events

Along with delegates, .NET offers a notion of event. Unlike the Event Library's approach, the model doesn't use ordinary objects for that purpose, but a special built-in concept of "event". It's a primitive of the object model, supported by language keywords: **event** in C#, **Event** in Visual Basic .NET, which you may use to declare special features (members) in a class.

What such a declaration introduces is actually not an event but an event type. Here too the .NET documentation doesn't try very hard to distinguish between types and instances, but for this discussion we have to be careful. When a .NET class Button in a GUI or WUI library declares a feature Click as **event** or **Event** it's because the feature represents the event type "mouse click", not a case of a user clicking a particular button.

In the basic event handling scheme, the declaration of any *event* type specifies an associated *delegate* type. That's how .NET enforces type checking for event handling. It all comes together in a type-safe way:

A • A delegate represents a routine.

B • That routine has a certain signature (number and types of arguments, and result if a function).

C • That signature determines a **delegate type**.

D • To define an **event type**, you associate it with such a delegate type.

E • Given the event type, any software element may *trigger* a corresponding event, passing to it a set of "event arguments" which must match the signature of the delegate. This matching is statically checkable; compilers for statically typed languages, and otherwise the type verifier of the .NET runtime, will reject the code if there is a discrepancy.

For step D, Visual Basic .NET lets you specify a routine signature without explicitly introducing a delegate type as you must do in C#. Internally the VB.NET compiler will generate a delegate type anyway. We'll restrict ourselves to the C# style, also possible in VB.NET. You can declare an event type as

```
public event AB_delegate Your_event_type ;                    /19/
```

where the last item is what's being declared; the delegate type that precedes, here AB_delegate, is an extra qualifier like **public** and **event**.

What this defines is a new event type such that triggering an event of type Your_event_type will produce event data of a type compatible with the signature of AB_delegate; in this example, the event will produce a value of type A and a value of type B. As another example you might declare

```
public event Two_coordinate_delegate Click ;
```

to declare an event type Click whose triggering will produce two integers corresponding to the mouse coordinates. This assumes (compare **/16/**):

```
public void delegate Two_coordinate_delegate (int x, int y);    /20/
```

Note again that a particular mouse click is an event, but Click itself denotes the general notion of a user clicking the mouse.

Such event type definitions are the counterpart of the declarations of instances of *EVENT_TYPE* in the Event Library. Here, however, we have to introduce a new type declaration, referring to a delegate type, in each case. The reason is clear: the .NET model has neither tuples nor genericity. In the Event Library we could declare for example

```
mouse_click: EVENT_TYPE [TUPLE [INTEGER, INTEGER]]
```

always using the same generic base class, *EVENT_TYPE*, and varying the generic parameter as needed. The equivalent of Your_event_type **/19/** would use *EVENT_TYPE* [*TUPLE* [*A*, *B*]]. In .NET, we have to introduce a new delegate type each time, and refer to it in declaring the event type.

## Connecting delegates to event types

We can now declare a delegate type corresponding to a certain routine signature; declare and create a delegate of that type, associated with a particular routine of matching signature; and define an event type that will generate event data also matching the delegate's type, so that the data can be processed by the routine. We need a facility enabling a subscriber to connect a certain delegate with a certain event type for a certain control.

We know the general idea: record a triple — control, event type, delegate — into an event-action table. In the .NET model and especially C#, however, the mechanism is expressed at a lower level. You will write

```
some_target.Some_event_type += some_delegate;
```

for example

```
your_button.Click += Two_coordinate_delegate;
```

where the highlighted += operator means "append to the end of a list". This appends the delegate to the list associated with the target and the event type. Such a list automatically exists whenever some_target is of a type that, among its features, has an event type Some_event_type. Then if a publisher triggers an event, the underlying .NET mechanisms will traverse all the lists associated with the event's type, and execute their delegates in sequence.

Collectively, the set of all such lists constitutes the equivalent of the event-action table. The subscriber, however, will have to know about the individual lists and manage them individually. The only operations permitted are += and its inverse –= which removes a delegate.

The introductory discussion of event-driven design pointed out that in the general case each association we record is not just a pair (event type, action) but a triple, involving a control. The .NET model's basic association mechanism this_control.this_event_type += this_action directly enforces this style. But then it doesn't cover the simpler case of subscribing to events that may occur anywhere, independent of any control. The Event Library gave us that flexibility: since an event type is just an object, you may create it at the level of the application **/1/**, or local to a particular control object such as a button **/5/**. You can achieve the former effect in .NET too, but this will require defining and creating a pseudo-control object and attach the event type to it, another source of extra code.

## Raising events

We haven't yet seen the publisher side. To raise an event of a type such as Your_event_type or Click you will, in C#, use the event type name as if it were a procedure name, passing the event data as arguments. Unfortunately, it is not enough to write Click (x_coordinate, y_coordinate) or, in the other example, Your_event_type (some_A_value, some_B_value). The proper idiom is

```
if (Click != null)
    {Click (this, x_mouse_coordinate, y_mouse_coordinate)};          /21/
```

The check for **null** is compulsory but can only be justified in reference to implementation internals. The name of the event type, here Click, denotes an internal list of delegates, to which you don't have access. (You can apply += and –=, but not to the whole list, only to the list for one control, such as your_button.Click.) If it's null because no delegate has been registered, trying to raise the event would cause an incorrect attempt to access a non-existent data structure.

It is incomprehensible that the mechanism makes this check incumbent on every publisher for every occurrence of raising an event. In the design of an API, one should avoid requiring client programmers to include an operation that's tedious to code and easy to forget, raising the risk that client applications will experience intermittent failures. Probably because of that risk, the documentation for .NET recommends never writing the above code **/21/** to raise an event directly, but encapsulating it instead in a routine

```
protected void OnClick (int x, int y)
    {if (Click != null)
        Click (this, x, y);
    }
```

then calling that routine to publish the event. The documentation actually says that you "*must*" use this style, including naming the routine On*EventType*, although in reality it's only a style recommendation. The strength of the advice indicates, however, the risks of not doing things right.

In comparing the .NET approach with the Event Library, we should also note that .NET's triggering of events requires some underlying magic. In the Event Library, the consequence of *mouse_click*.*publish* (*x*, *y*) is to execute the procedure *publish* of class *EVENT_TYPE*. That procedure looks up the event-action table to find all agents that have been associated with the *mouse_click* event type, an ordinary object. The table implementation is a plain *HASH_TABLE* from the standard EiffelBase library. You can see the whole implementation (only if you want to, of course) by looking up the source code of *EVENT_TYPE*. If you are not happy with the implementation you can write a descendant of *EVENT_TYPE* that will use different data structures and algorithms. In .NET, however, you have to accept that the instruction to register an event type, such as your_button.Click += your_delegate, somehow updates not only the list Click attached to your_button but also the mysterious global list Click which you can't manipulate directly, and in fact shouldn't have to know about except that you must test it against **null** anyway. So you have to trust the .NET runtime to perform some of the essential operations for you. Of course the runtime probably *deserves* to be trusted on that count, but there is really no reason for such mysteries. Implementing the action-event table is plain application-level programming that doesn't involve any system-level secrets and should have been done in a library, not in the built-in mechanisms of the virtual machine.

## Event argument classes

If you have been monitoring the amount of extra code that various rules successively add to the basic ideas, you have one more step to go. For simplicity, our example event types, Click and Your_event_type, have relied on delegate types by specifying signatures directly: two integer arguments for Click **/20/**, arguments of types A and B in the other case **/16/**.

The recommended style is different. (Here too, the documentation suggests that this style is required, whereas it's in fact just a possible methodological convention.) It requires any delegate type used with events — meaning, really, any delegate type — to have exactly two arguments, the first one an Object representing the target and the second a descendant of the library class System.EventArgs. In our examples, you would declare

```
                                                    // Compare with /16/:
public void delegate AB_delegate (Object sender, ABEventArgs e)

                                                    // Compare with /20/:
public void delegate Two_coordinate_delegate (Object sender,
                                                    TwoCoordinateEventArgs e)
```

with special classes ABEventArgs, describing pairs of elements of types A and B, and TwoCoordinateEventArgs describing pairs of integer elements representing mouse coordinates. For an event that doesn't generate event data, you would use the most general class System.EventArgs.

The reason for this rule seems to be a desire to institute a fixed style for all event handling, where events take exactly two arguments, the details of the event data being encapsulated into a specific class.

The consequence, however, is an explosion of small classes with no interesting features other than their fields, such as mouse coordinates *x* and *y*. In ordinary object-oriented methodology, the proliferation of such classes — really record types in O-O clothing — is often a sign of trouble with the design. It is ironic that mechanisms such as delegates succeed in countering a similar proliferation, arising from the presence of many "command classes", used for example in the Undo-Redo pattern **[10]**. Reintroducing numerous System.EventArgs classes, one per applicable signature, is a step backwards.

The resulting style is particularly heavy in the case of events with *no* data: instead of omitting arguments altogether, you must still pass **this** (the current object) as Sender, and a useless object of type System.EventArgs — an event data object containing no event data. Such overkill is hard to justify.

> In the absence of compelling arguments for the System.EventArgs style, it would seem appropriate to advise .NET event programmers to disregard the official methodological rule. The style, however, is used heavily in the event-driven libraries of .NET, especially Windows Forms for GUI design and Web Forms for WUI design, so it's probably there to stay.

## An assessment of the .NET form of event-driven design

The event-delegate mechanism of .NET clearly permits an event-driven style of design. It lies at the basis of Windows Forms and Web Forms, both important and attractive libraries in the .NET framework. We must keep this in mind when assessing the details; in particular, none of the limitations and complications encountered justifies returning to a Java approach where the absence of a mechanism to wrap routines in objects causes even more heaviness.

The amount of noise code is, however, regrettable. Let's recapitulate it by restarting from the Event Library model, which seems to yield the necessary functionality with the conceptually minimal notational baggage. On the publisher side we must, returning to the mouse click example:

E1 • Declare an event type *click*: *EVENT_TYPE* [*TUPLE* [*INTEGER, INTEGER*]] typically as a once function that creates the object.

E2 • For every occurrence of the event, publish it: *click*.*publish* (*x, y*).

On the subscriber side, to subscribe a routine *r*, we execute, once:

E3 • *your_button*.*click*.*subscribe* (**agent** *r*)

Here is the equivalent in .NET, again using C#. Some element of the system (it can be the publisher or the subscriber) must:

D1 • Introduce a new descendant ClickArgs of EventArgs repeating the types of arguments of *r*. This adds a class to the system.

D2 • Declare a delegate type ClickDelegate based on that class. This adds a type.

Then the publisher must:

D3 • Declare a new event type ClickEvent based on the type ClickDelegate. This adds a type.

D4 • Introduce a procedure OnClick able to trigger a Click event, but protecting it against the **null** case. The scheme for this is always the same, but must be repeated for every event type.

D5 • For every occurrence of the event, create an instance of the ClickArgs class, passing x and y to its constructor. This adds a run-time object.

D6 • Also for every occurrence of the event, call Click with the newly created object as argument.

The subscriber must, to subscribe a routine *r*:

D7 • Declare a delegate rDelegate of type ClickDelegate.

D8 • Instantiate that delegate with *r* as argument to the constructor (this step can, in C#, be included with the previous step as a single declaration-initialization, see **/17/**).

D9 • Add it to the delegate list for the event type, through an instruction of the form your_button.Click += rDelegate.

In the case of an event type that is not specific to a control, it is also necessary, as we have seen, to add a class describing an artificial control. (With the Event Library you just *remove* the control, *your_button*, from E3.)

To all this encumbrance one must add the consequences of the delegate mechanism's lack of support for closed arguments and open targets, as permitted by agents. These limitations mean that it is less frequently possible, starting from an existing application, to reuse one of its routines directly and plug it into an event-driven scheme, for example a GUI or WUI. If the argument signature is just a little different, you will need to write new wrapper routines simply to rearrange the arguments. More glue code.

← *"Avoiding glue", page 11*..

The combination of these observations explains why examples of typical event-driven code that would use a few lines with the Event Library can extend across many pages in .NET and C# books.

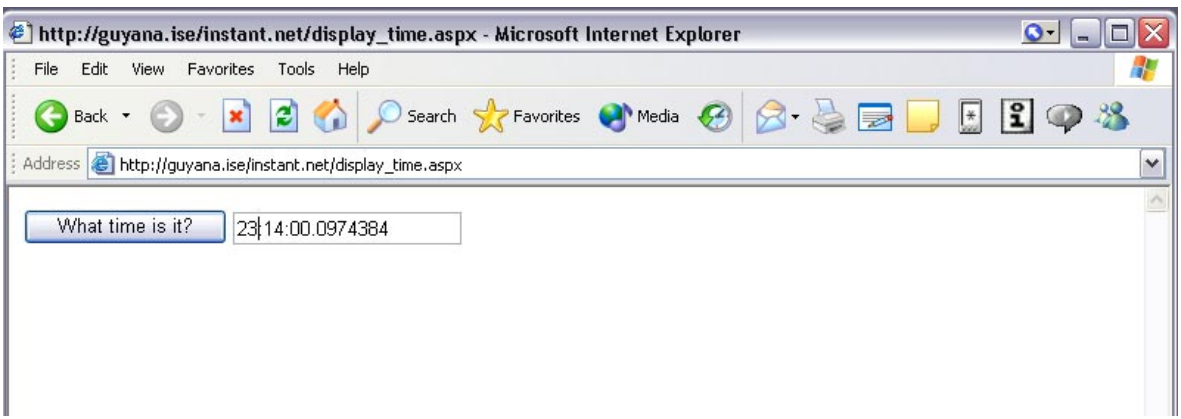
This does not refute the observation that .NET essentially provides the needed support for event-driven design. The final assessment, however, is that the practical use of these mechanisms is more complicated and confusing than it should be.

# 6 EVENTS FOR WEB INTERFACES IN ASP.NET

As a complement to the preceding discussion of .NET delegates and events it is interesting to note that the .NET framework does offer a simple and easy-to-learn mechanism for building event-driven applications. It's not part of the basic Common Language Runtime capabilities, but rather belongs to the application side, in the Web Forms mechanism of ASP.NET. Internally, it relies on the more complex facilities that we have just reviewed, but it provides a simpler set of user facilities. The "users" here are not necessarily programmers but possibly Web designers who will, from Web pages, provide connections to an underlying application. So the scope is far more limited — ASP.NET is a web development platform, not a general programming model — but still interesting for this discussion.

ASP.NET is a set of server-side tools to build "smart" Web pages, which provide not only presentation (graphics, HTML) but also a direct connection to computational facilities implemented on the Web server. Because ASP.NET is closely integrated with the rest of the .NET framework, these facilities can involve any software implemented on .NET, from database operations to arbitrarily complex programs written in any .NET-supported language such as C#, Visual Basic or Eiffel.

An event-driven model presents itself naturally to execute such server operations in response to choices — button click, menu entry selection… — executed by a visitor to the Web page. The screen shown at the beginning of this article was indeed extracted from an ASP.NET application:



If the visitor clicks What time is it?, the current time of day appears in the adjacent text field.

The code to achieve this is, even using C#, about as simple as might be hoped. The whole text reads:

```
<html>
    <body>
        <form runat="server">
            <asp:Button OnClick = "display_time"
            Text = "What time is it?" runat = server/>
            <asp:TextBox id = "Output" runat=server/>
        </form>
    </body>

    <script language="C#" runat=server>
        private void display_time (object sender, EventArgs e)
            {Output.Text= DateTime.Now.TimeOfDay.ToString();}
    </script>
</html>
```

The first part, the <body>, describes the layout of the page and the second part, <script>, provides the associated program elements, here in C#.

The <body> describes two ASP.NET controls: a button, and a text box called **Output**. ASP.NET requires making them part of a <form> to be runat the server side. The first highlighted directive sets the OnClick attribute of the button to **display_time**, the name of a procedure that appears in the <script> part. That's enough to establish the connection: when a Click event occurs, the procedure **display_time** will be executed.

The <script> part is C# code consisting of a single procedure, **display_time**, which computes the current time and uses it to set the Text property of the **Output** box.

This does what we want: a Click event occurring in the button causes execution of **display_time**, which displays the current time in the **Output** box.

> The time computation uses class DateTime, where feature Now gives the current date and time, of type Date; feature TimeOfDay, in Date, extracts the current time; and ToString produces a string representation of that time, so that we can assign it to the Text feature of the TextBox.

Such simplicity is possible because ASP.NET takes care of the details. Since ASP.NET knows about the Click event type, controls such as asp:Button include an OnClick property, which you can set to refer to a particular procedure, here **display_time**. As a result, we don't see any explicit delegate in the above code; .NET finds the name of the procedure **display_time**, and takes care of generating the corresponding delegates.

The only hint that this involves delegates is in the signature of **display_time**, which involves two arguments: sender, of type object, and e, of type EventArgs. In the recommended style, as we have seen, they are the arguments representing event data, which delegate methods are expected to handle. Someone who just learns ASP.NET without getting the big picture will be told that (object sender, EventArgs e) is a magic formula to be recited at the right time so that things will work.

Clearly, the .NET machinery translates all this into the standard event-delegate mechanism discussed in the previous section. But it is interesting to note that, when the target audience is presumed less technical — Web designers rather than hard-core programmers — .NET can offer a simple and clear API.

The Event Library provides similar simplicity in a more general programming model.

# 7 EVENT LIBRARY COMPLEMENTS

The essential properties of the Event Library were given at the beginning of this discussion. Here are a few complementary aspects, to indicate perspectives for more specialized uses.

← *"EVENT LIBRARY ESSENTIALS", 2, page 2*..

The library is available for free download, in source form, from se.inf.ethz.ch. Another reference **[2]** provides more details.

## Basic features

First let's examine some uses of the class and of the two features already introduced. The class is declared as

```
EVENT_TYPE [EVENT_DATA –> TUPLE]
```

meaning that the generic parameter represents an arbitrary tuple type. The two basic features, as we have seen, are *publish* and *subscribe*.

To introduce an event type you simply declare an entity, say *your_event_type*, with the type *EVENT_TYPE* [*TUPLE* [*A*, *B*, …]] for some types *A*, *B*, …, indicating that each occurrence will produce event data containing a value of type *A*, a value of type *B* and so on. If there is no event data use an empty tuple type as parameter: *EVENT_TYPE* [*TUPLE* [ ]].

*your_event_type* will denote an ordinary object — an instance of *EVENT_TYPE* — and you may declare it using any of the generally available techniques. One possibility, as we have seen, is to make it a "once function" so that it will denote a single run-time object, created the first time any part of the software requests it. You may also attach it to every instance of a certain class representing a control, for example as a "once per object" function. Many other variants are possible.

The two basic procedures have the signatures

```
subscribe (action: PROCEDURE [ANY, EVENT_DATA])

publish (args: EVENT_DATA)
```

The type *PROCEDURE* [*G, H*], from the Kernel Library, describes agents built from any procedure declared in a descendant class of *G* and taking arguments conforming — when grouped into a tuple — to *H*, a tuple type. Here this means that for *EVENT_TYPE* [*TUPLE* [*A, B, …*]] you may *subscribe* any procedure, from any class, that takes arguments of types *A, B, ….* (For details of agents see **[4]** or **[11]**.)

The procedure *publish* takes an argument of type *TUPLE* [*A, B, …*] — a sequence of values such as [*some_a_value, some_b_value*] denoting a tuple — enabling a publisher to trigger an event with appropriate event data through

```
your_event_type.publish ([some_a_value, some_b_value])
```

as in the earlier example **/3/**.

## Introducing specific event types

If for a certain event type you know the exact event data constituents, you can avoid using tuples by defining a specific heir of *EVENT_TYPE*. You might use this technique to cover a category of mouse events including left click, mouse click, right click, control-right-click, mouse movement etc. which all produce event data of a type *MOUSE_POSITION* represented by an existing class such as

```
class
    MOUSE_POSITION
feature -- Access
    x: INTEGER
        -- Horizontal position

    y: INTEGER
        -- Vertical position

    … Other features (procedures in particular) …
end
```

Publishers of mouse events might have access to an object *current_position* of type *MOUSE_POSITION* representing the current mouse coordinates. By default they would trigger an event through calls such as

```
control_right_click.publish ([current_position.x, current_position.y])  /22/
```

But it may be more convenient to let them use the object *current_position* directly. If you define

```
class
    MOUSE_EVENT_TYPE
inherit
        EVENT_TYPE [TUPLE [INTEGER, INTEGER]]
feature -- Basic operations
    publish_position (p: MOUSE_POSITION) is
            -- Trigger event with coordinates of p.
        do
            publish ([p.x, p.y])
        end
end
```

you can use this class rather than *EVENT_TYPE* to declare the relevant event types such as *control_right_click*, *left_click*, *mouse_movement* and so on, then publish mouse events, instead of **/22/**, through

```
control_right_click.publish_position (current_position)
```

or even, if you give class *MOUSE_EVENT_TYPE* access to *current_position*, enabling it to include a procedure *publish_current_position* with no argument, through just

```
control_right_click.publish_current_position
```

Classes such as *MOUSE_EVENT_TYPE* correspond, in the .NET model, to specific descendants of System.EventArgs. The difference is that you are not required to define such classes; you'll introduce one only if you identify an important category of event types with a specific form of event data, globally captured by an existing class, and — as a convenience — want to publish the event data as a single object rather than as a tuple. In all other cases you'll just use *EVENT_TYPE* directly, with tuples. So you avoid the proliferation of useless little classes, observing instead the object technology principle that enjoins to add a class to a system only if it represents a significant new abstraction.

## Subscriber precedence

If several agents are subscribed to an event type, the associated routines will by default be executed, when events occur, in the order of their subscription. To change this policy you may directly access the list of subscribers and change the order of its elements. To facilitate this, class *EVENT_TYPE* is a descendant of *LIST* [*ROUTINE* [*ANY, TUPLE*[]], so that all the operations of the EiffelBase class *LIST* are applicable.

> Although such uses of inheritance are appropriate — see the detailed discussion in **[10]** — they run contrary to some commonly held views on O-O design methodology. It would be possible to make *EVENT_TYPE* a client rather than heir of *LIST* by including a feature *subscribers* of type *LIST* [*ROUTINE* [*ANY, TUPLE*[]]; although making the class less easy to use, this change would not affect the rest of this discussion.

The principal factor in this decision to provide access to the list of subscribers was successful user experience with the EiffelVision 2 multi-platform GUI library **[5]**, which follows the same convention. Reusing the EiffelBase list structures gives clients a whole wealth of list operations; the default *subscribe* is an *extend*, the operation that adds an element at the end of a list, but you can also use all the traversal operations, *replace* to replace a particular element, *put_front* to add an element at the beginning, the list deletion operations and others. This differs from .NET approach:

- In .NET, you *have* to consider an event type as a list. Here you can just use *subscribe* without bothering or knowing about lists. Only for advanced uses that need fine control over the subscriber list will you start manipulating it directly.

- In .NET, as noted, the list is always local to a control. Here it's just a standard list object and may appear anywhere in the software structure, at the level of the application or local to another object.

- The delegate lists associated with .NET events are very special structures, with only two applicable operations, += and –=. Here, they are general lists, to which you can apply all the EiffelBase list features.

## Ignoring some publishers

Subscribers can be selective about an event's originating publisher. By default an event will cause the execution of subscribed routines regardless of the publisher. Call *ignore_publisher* (*p*: *ANY*) to exclude from consideration any event triggered by *p*. To ignore events except if they come from specific publishers, use *consider_only* (*p*: *ARRAY* [*ANY*]). To include a specific publisher explicitly, use *consider_publisher* (*p*: *ANY*). To cancel all subscriptions, use *ignore_all_publishers*; to reset to the default policy, use *consider_all_publishers*. These procedures are cumulative: a call to any of them complements or overrides the policy set by previous calls. To find out the resulting status you may use *ignored_publishers* and *considered_publishers*, both queries returning an *ARRAY* [*ANY*], the later meaningful only if the boolean query *specific_publishers_only* returns True.

To ignore all events temporarily and start considering them again, use *suspend_subscription* and *restore_subscription*. Unlike the *ignore* and *consider* variants these do not make any permanent change to the set of considered publishers.

This last set of facilities lends itself to criticism of pointless "featurism". As noted, however, such extra functionality does not harm simple uses of the library. It may have to be adapted or removed in the future.

# 8 SOFTWARE ENGINEERING LESSONS

The first goal of this presentation has been immediate and pragmatic: to present the Event Library as a general tool for building event-driven software solutions. We may also draw some more general observations.

## Limitations

The Event Library — like the other approaches surveyed — has so far been typically used for sequential or quasi-sequential applications, such as GUI development. Although its default ordering semantics is clear (procedures subscribed to the same event type will be executed, when an event occurs, in the order in which they have been subscribed), a generalization to full parallel computation would require precise rules on synchronization in the case of concurrent events.

In addition, the presentation of the mechanisms has not included any discussion of correctness; such a discussion should be based on the contracts associated with the routines that we encapsulate in agents.

Concurrency and correctness issues are both clear candidates for further extensions of this work.

## The virtue of simplicity

The word “simple” has occurred more than a few times in this presentation. Although claiming that something is simple doesn’t make it so, we hope that the reader will have noticed the small number of concepts involved in using the Event Library.

This concern for simplicity applies not only to the library but also to the underlying language design, which attempts to maximize the “signal-to-noise ratio” both by providing powerful constructs, such as agents, and minimizing the noise by avoiding the provision of two solutions wherever one will do.

Although it may be tempting to dismiss the search for simplicity as a purely esthetic concern, the results seem clear when we compare the effort it takes for an ordinary user — an application builder who wants to use an event-driven library — to implement the Observer Pattern or use the .NET mechanisms rather than relying on the Event Library.

*← The steps involved were listed in “An assessment of the .NET form of event-driven design”, page 29..*

## The search for abstractions

The key to the Event Library's simplicity is in the choice of its basic abstraction. Previous solutions used different ones:

- The Observer Pattern relies on abstractions "Subject" and "Observer" which, however intuitive, are not particularly useful since they have no relevant features.
- The .NET model produces complicated programs because it insists on defining a new delegate type — a new abstraction — for every routine signature to be used in handling events, and a new event type — again a new abstraction, with its own name — for every kind of event that a system may have to process. In the example discussed, it requires a new delegate type for procedures that take two integer arguments (where the Event Library simply uses an Eiffel agent expression, relying on genericity to ensure typing) and a new class to describe the mouse click event type. This leads to name and code explosion.

Both cases seem to result from what has been called "taxomania" **[10]**: overuse of inheritance and introduction of useless classes.

The Event Library identifies, as its key abstraction, the single notion of event type, characterized — as any proper abstraction in the abstract-data-type view at the basis of object technology — by relevant features: a subscriber can *subscribe* to be notified of events of a given type, and a publisher can *publish* an event of a given type. This choice of abstraction makes all the difference.

## Lines of research

This discussion has emphasized a certain path to software construction where the main task is to search for the right abstractions.

Such emphasis may seem remote from the concepts that seem currently to dominate discussions of design methodology, from UML and Aspect-Oriented Programming to Extreme Programming and Agile Methods. Indeed there is nothing new in the idea of identifying the problem's abstractions. But in light of the tendency to take object technology for granted **[12]**, it is useful to note that resolving design problems may follow not just from new techniques but from the creative application of known principles. Good O-O design requires going back repeatedly to the basic question: "What *are* the main data abstractions behind this problem?".

## From patterns to components

One of the design concepts that has amply proved its usefulness is the idea of design patterns. The patterns work has been instrumental in helping to identify and classify important algorithmic and architectural schemes.

When assessed against *reuse* goals, however, design patterns seem to go the other way, possibly contradicting some of the ideas that have made object technology attractive. Patterns are techniques that developers must learn and implement, like repertoires of traditional algorithms and data structures that one learns as a student and applies as a programmer. Coming aftter object and component technology, patterns seem to imply a return to recipes which, however elegant, must be applied afresh by every developer in every project.

This view that reusing packaged components is preferable to repeatedly handcrafting specific solutions leads to what we may call the *Pattern Elimination Conjecture*: that in the long term any useful pattern should be discarded as a pattern, and replaced by reusable components with a clear, simple, directly usable interface. Here the originators of the pattern idea would respond that typical patterns are too sophisticated to be encapsulated into components; but then the conjecture would assert that this difficulty of going from patterns to components is due to two factors:

- The limitations of the programming languages that served to describe the original patterns. (The “Prototype” pattern, for example, disappears as a pattern in Eiffel to become a direct application of the Kernel Library’s built-in cloning mechanism.)

- Possibly insufficient effort or insight in previous attempts to turn patterns into components.

The Pattern Elimination Conjecture implies no criticism of the pattern idea; to the contrary, its recognizes the essential contribution of patterns to identifying the right components, part of what the reuse literature calls “domain analysis”. It states that the natural goal for a pattern, once identified, is to cease being a pattern and become packaged as a component. As noted in **[10]**, it’s a natural ambition for object technology to make any statement of the form “*X* considered harmful” self-fulfilling — as soon as *X*, whatever it is, has been proved *useful* — by the simple observation that if it’s useful it should be componentized.

This article's analysis of the Observer pattern and its introduction of the Event Library confirm the Pattern Elimination Conjecture in one case, since it's easier to use the library than to learn the architecture of the Observer pattern and apply it to a new program. This is of course just one example. Work by Karine Arnout and the author (see **[1]**) is currently exploring whether this partial result can be generalized to other important design patterns as described in the literature.

## Types and instances

It has been noted **[10]** **[13]** that publications on object technology too often use the terms "class" and "object" for one another. The prevalence of this confusion is surprising, as there is nothing difficult here: a class is a model, an object is an instance of such a model; classes exist in the software text, objects existing in memory at execution time. Yet one continues to read about a program design that includes "an Employee object" and other such nonsense.

Some would argue that insisting on the distinction is just being fussy, and that readers understand what is meant in each case. Without having an absolute way to know, we may recall the lack of any attempt, in the .NET documentation, to distinguish between event types and events, or between delegate types and delegates, and conjecture that a more careful approach might have led to a different choice of basic abstraction and to a mechanism easier to understand and use.

## Programming language constructs

Our final observation addresses the role of programming languages. The solution applied by the Event Library is made possible by the combination of a number of language features beyond the basic object-oriented concepts:

- Genericity (the key to avoiding a proliferation of little event and delegate classes), which a satisfactory object model should include in addition to inheritance **[9]**.
- Tuple types (also important for this purpose, thanks to their support for variable-length sequences).
- *Constrained* genericity (used here to ensure that certain generic arguments can only represent tuples).
- Agents (and their typing properties).
- The possibility of using open as well as closed arguments in an agent, and of keeping the target open if desired (the key to avoiding a proliferation of glue code).

← *Through the notation* EVENT_DATA–>TUPLE *in the declaration of* EVENT_TYPE, *page* 36.

- Once functions (taking care of the problems of object initialization and sharing without breaking the decentralization of object-oriented architectures).

- Multiple inheritance (essential in particular to the structure of the underlying EiffelBase library).

- Covariance (for the needed type flexibility, in spite of the associated type checking issues).

With the growing acceptance of object-oriented ideas as a basis for new languages, there may be a tendency to assume that all O-O models are essentially equivalent. They are not. The features listed, many of them not supported by most O-O languages, make the difference in the ease of use, extendibility and reusability of the solutions encountered in this discussion.

## Scope

Event-driven design is attractive in a number of situations illustrated by the examples of this article. It would be useful to conclude with a precise analysis of how it relates to other design styles, and how wide a range of applications it encompasses. We can, however, offer no firm answer on either count.

The most natural comparison is with concurrent computation mechanisms. Event-driven design indeed assumes some concurrency between the publishers and the subscribers, but that concurrency remains implicit in the model. Analogies that come to mind are with CSP **[7]**, with its input and output events, and with the Linda approach to concurrent computation **[3]** whose general scheme involves clients depositing computational requests into a general "tuple space" which suppliers then retrieve and process based on pattern matching. We have not, however, explored such analogies further. Some work, already noted **[18]**, is intended to *replace* event-driven design by concurrent computation mechanisms.

We are also not able to provide a clear assessment of the scope of the design style presented in this discussion. It undoubtedly works well in its usual areas of application, mainly GUI and now WUI building. How general is the idea, illustrated in an earlier figure, of publishers throwing events like bottled messages into the ocean, with the hope that some subscriber will pick them up? It may be a powerful paradigm that can affect the structure of many systems, not just their relation to their user interfaces; or maybe not.

← *Page 7.*

On one issue, language-related, we now have unambiguous evidence: the usefulness of equipping an object-oriented language with a way to encapsulate routines into objects, such as Eiffel's agents or the delegate facility of .NET. The introduction of agents initially raised concern that they might in certain cases compete with the more traditional O-O constructs. Extensive experience with the mechanism has dispelled this concern; agents have a precise place in the object-oriented scheme, and in practice there is no ambiguity as to where they should be used and where not. The library-based scheme of event-driven computation described in this article is a clear example of when agents can provide an indispensable service.

## Acknowledgments

This article rests on the work of the people who designed the agent mechanism: Paul Dubois, Mark Howard, Emmanuel Stapf and Michael Schweitzer. It also benefits from the design of the EiffelVision 2 library and its use of agents and event-driven mechanisms, due among others to Sam O'Connor, Emmanuel Stapf, Julian Rogers and Ian King. It takes advantage of comments from Karine Arnout and Volkan Arslan. It is indebted to the other designs discussed, especially Smalltalk's MVC, the Observer Pattern, the .NET event model, and its realization in C# and VB.NET. Without implying endorsement of the ideas expressed I gratefully acknowledge the comments received on earlier versions of this work from Éric Bezault, Jean-Marc Jézéquel, Piotr Nienaltowski, Claude Petitpierre and a referee who, when my request was granted to lift anonymity in light of the value of his criticism, turned out to be Tony Hoare.

## References

[1] Karine Arnout, *Contracts and Tests*, research plan at se.inf.ethz.ch/people/arnout/phd_research_plan.pdf, consulted June 2003.

[2] Volkan Arslan, Piotr Nienaltowski and Karine Arnout. *Event library: an object-oriented library for event-driven design*, to appear in JMLC 2003, Proceedings of Joint Modular Languages Conference, Klagenfurt (Austria), August 2003, ed. Laszlo Böszörmenyi, Springer-Verlag, 2003.

[3] Nicholas Carriero and David Gelernter: *How to Write Parallel Programs*, MIT Press, 1990. More recent (2000) Linda tutorial at lindaspaces.com/downloads/lindamanual.pdf, consulted June 2003.

[4] Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer and Emmanuel Stapf: *From Calls to Agents*, in *Journal of Object-Oriented Programming*, vol. 12, no. 6, June 1999.

[5] Eiffel Software: *Online EiffelVision2 documentation* at docs.eiffel.com/libraries/vision2/, consulted June 2003.

[6] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] C.A.R. Hoare: *Communicating Sequential Processes*, Prentice Hall, 1985.

[8] Jean-Marc Jézéquel, Michel Train and Christine Mingins: *Design Patterns and Contracts*, Addison-Wesley, 1999.

[9] Bertrand Meyer: *Genericity versus Inheritance*, in Norman K. Meyrowitz (Ed.): Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings. SIGPLAN Notices 21(11), November 1986, pages 391-405. Updated version in appendix B of **[10]**.

[10] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[11] Bertrand Meyer, Agent chapter in *Eiffel*: *The Language, 3rd edition*, in preparation, chapter text online at www.inf.ethz.ch/personal/meyer/publications/, consulted June 2003.

[12] Bertrand Meyer: *A Really Good Idea* (final installment of Object Technology column), IEEE *Computer*, vol. 32, no. 12, December 1999, pages 144-147. Also online at www.inf.ethz.ch/personal/meyer/publications/, consulted June 2003.

[13] Bertrand Meyer: *Assessing a C++ Text* (review of *Programming C#* by Jesse Liberty), IEEE *Computer*, vol. 35, no. 4, April 2002, pages 86-88. Also online at www.inf.ethz.ch/personal/meyer/publications/, consulted June 2003.

[14] Bertrand Meyer: *Multi-Language Programming*; *How .NET Does It*, published in three parts in *Software Development* Magazine, "Beyond Objects" column, May-July 2002. Also online at www.inf.ethz.ch/personal/meyer/publications/, consulted June 2003.

[15] Bertrand Meyer, Raphaël Simon, Emmanuel Stapf: *Instant .NET*, Prentice Hall, 2004, in preparation.

[16] *.NET Framework Class Library: Delegate Class*, part of Microsoft .NET documentation included with the .NET framework, also online at msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemDelegateClassTopic.asp, consulted June 2003.

[17] *Visual Basic Language Reference: Delegate Statement*, part of Microsoft .NET documentation included with the .NET framework, also online at msdn.microsoft.com/library/default.asp?url=/library/en-us/vblr7/html/vastmDelegate.asp, consulted June 2003.

[18] Claude Petitpierre, *A Design Pattern for Interactive Applications*, École Polytechnique Fédérale de Lausanne, 2002.

[19] Sun Microsystems: *About Microsoft's "Delegates"*, 1997 white paper online at java.sun.com/docs/white/delegates.html, consulted June 2003.