Event Library: an object-oriented library for event-driven design

Volkan Arslan, Piotr Nienaltowski, Karine Arnout

Swiss Federal Institute of Technology (ETH), Chair of Software Engineering,
8092 Zurich, Switzerland
{Volkan.Arslan, Piotr.Nienaltowski,
Karine.Arnout}@inf.ethz.ch
http://se.inf.ethz.ch

Abstract. The Event Library is a simple library that provides a solution to the common problems in event-driven programming. Its careful design facilitates further extensions, in order to satisfy users' advanced needs. It has been implemented in Eiffel, taking advantage of the advanced mechanisms of this language. In this paper, we present the architecture of the library and show how it can be used for building event-driven software. The discussion is illustrated with a sample application.

1 Introduction

Event-driven programming has gained considerable popularity in the software engineering world over the past few years. Several systems based on this approach have been developed, in particular Graphical User Interface (GUI) applications. Event-driven techniques facilitate the separation of concerns: the application layer (also called *business logic*) provides the operations to execute, whereas the GUI layer triggers their execution in response to human users' actions.

Similar ideas have been proposed under the names *Publish/Subscribe* and *Observer pattern*. The latter, introduced in [6], purports to provide an elegant way to building event-driven systems. It certainly contains useful guidelines for the development of event-driven programs; however, it falls short when talking about reuse: developers have to implement the pattern anew for each application. In order to solve this problem we decided to turn the Observer pattern into an Eiffel library, so that it can be reused without additional programming effort.

Despite its small size — just one class — the Event Library is powerful enough to implement the most common event-driven techniques, and it can be extended to handle users' advanced needs. Its simplicity results from using specific mechanisms of Eiffel, including Design by Contract™, constrained genericity, multiple inheritance, agents, and tuples. The underlying concepts of event-driven programming are presented in [9]. The same article provides also a review of existing techniques, such as the .NET delegate mechanism and "Web Forms". In this paper, we focus on the Event Library.

The rest of the paper is organized in the following way. Section 2 explains how to use the Event Library, and illustrates it with an example. Section 3 describes the architecture of the Event Library and shows how it fulfills the *space*-, *time*-, and *flow-decoupling* requirements of the Publish/Subscribe paradigm. Section 4 draws conclusions and discusses possible extensions of the library.

2 Event Library in Action

We use a sample application to show the basic capabilities of the Event Library. We explain, step by step, how to use the Event Library to build an event-driven application. Both the library and the example presented here are available from [1]. They were developed with the EiffelStudio 5.2 graphical development environment [3], and they can be used on any platform supported by EiffelStudio[1].

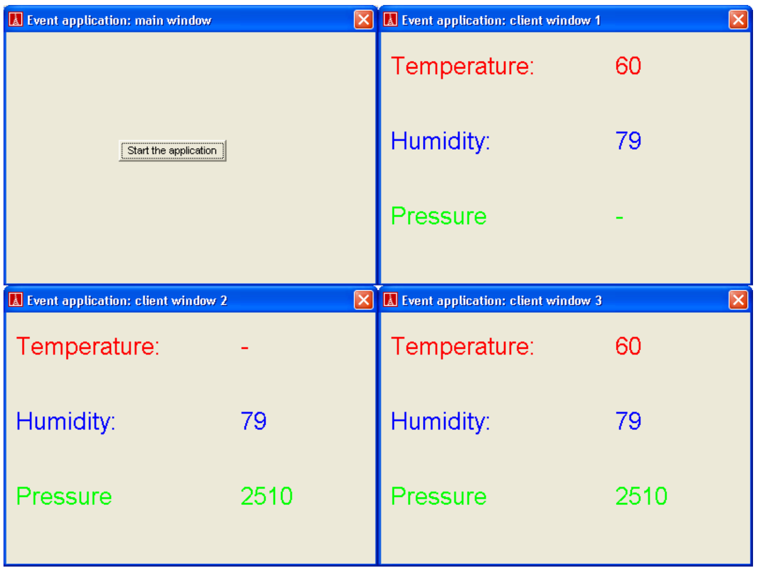

Fig. 1. Sample Event application

2.1 Sample Event-Driven Application

We want to observe the temperature, humidity, and pressure of containers in a chemical plant. We assume that the measurement is done by external sensors. Whenever the

[1] Microsoft Windows, Linux, Unix, VMS, Solaris

value of one or more measured physical attributes changes, the concerned parts of our system (e.g. display units) should be notified, so that they can update the values.

There are several reasons for choosing an event-driven architecture for such application. First of all, we should take into account the event-driven nature of the problem: input values are coming from external sensors at unpredictable moments, and the application is reacting to their change. Secondly, the independence between the GUI and the "business logic" is preserved. If the physical setup changes (e.g. sensors are replaced by different ones, new display units are introduced), the system can be easily adapted, without the need to rewrite the whole application.

Compiling and launching the sample causes four windows to appear (Fig. 1). The top-left window is the main application window. It displays information about the subsequent execution phases of the application. Three client windows display the values of temperature, humidity, and pressure in a chemical container. Note that these three display units correspond to the same container; however, they are interested in different sets of measures: Client window 1 shows temperature and humidity; Client window 2 shows humidity and pressure; Client window 3 shows temperature, humidity, and pressure. Each of these windows can change its "interests" over time, either by subscribing to a given event type (see 2.5) or by unsubscribing from it (see 2.6). All subscriptions may be also temporarily suspended (see 2.7).

2.2 Using the Event Library

Fig. 2 shows the overall architecture of our sample application. BON notation [11] is used. The arrows represent the client-supplier relationship.

The application is divided into three clusters: `event`, `application`, and `gui`. The `event` cluster contains one class, `EVENT_TYPE`, which abstracts the general notion of an *event type*. The `application` cluster contains the application-specific class `SENSOR`. The `gui` cluster groups GUI-related classes, including `MAIN_WINDOW`, `APPLICATION_WINDOW`, and `START`.

Class `SENSOR` models the physical sensor that measures temperature, humidity, and pressure. The class contains the corresponding three attributes: `temperature`, `humidity`, and `pressure`, used for recording values read on the physical sensor.

```
class SENSOR
  feature -- Access
    temperature: INTEGER
         -- Container temperature
    humidity: INTEGER
         -- Container humidity
    pressure: INTEGER
         -- Container pressure
end
```

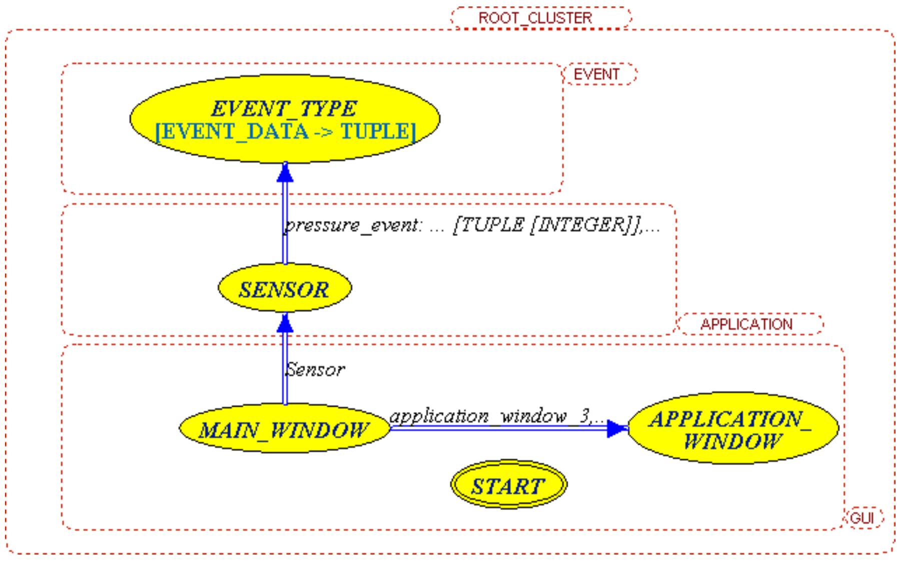

Fig. 2. Class diagram of the sample application

A `set` feature is provided for each attribute, e.g.

```
set_temperature (a_temperature: INTEGER) is
        -- Set temperature to a_temperature.
  require
    valid_temperature:
      a_temperature > -100 and a_temperature < 1000
  do
    temperature := a_temperature
  ensure
    temperature_set: temperature = a_temperature
  end
```

Note the use of assertions — preconditions and postconditions — to ensure correctness conditions. The *precondition* states that the temperature read by the sensor must be between -100° and 1000°; the *postcondition* ensures that the temperature is equal to the temperature read by the sensor.[2]

2.3 Creating an Event Type

We need to define an event type corresponding to the change of attribute `temperature` in class `SENSOR`; let us call it `temperature_event`:

[2] For the purpose of the subsequent discussion, in particular in code examples, we will only use the attribute `temperature`. Similar code is provided for attributes `humidity` and `pressure`, although it does not appear in the article.

```
feature -- Events
  temperature_event: EVENT_TYPE [TUPLE [INTEGER]]
    -- Event associated with attribute temperature
invariant
  temperature_event_not_void: temperature_event /= Void
```

To define the temperature event, we use the class EVENT_TYPE (from the event cluster), declared as EVENT_TYPE [EVENT_DATA -> TUPLE]. It is a generic class with constrained generic parameter EVENT_DATA representing a tuple of arbitrary types. In the case of temperature_event, generic parameter is of type TUPLE [INTEGER] since the event data (temperature value) is of type INTEGER.[3]

2.4 Publishing an Event

After having declared temperature_event in class SENSOR, we should make sure that the corresponding event is published when the temperature changes. Feature set_temperature of class SENSOR is extended for this purpose:

```
set_temperature (a_temperature: INTEGER) is
    -- Set temperature to a_temperature.
    -- Publish the change of temperature.
  require
    valid_temperature:
      a_temperature > -100 and a_temperature < 1000
  do
    temperature := a_temperature
    temperature_event.publish ([temperature])
  ensure
    temperature_set: temperature = a_temperature
  end
```

The extension consists in calling publish with argument [temperature] (corresponding to the new temperature value) on temperature_event. Class SENSOR is the *publisher* of the temperature event.

2.5 Subscribing to an Event Type

We need to complete our sample application with other classes that will subscribe to events published by SENSOR. First, we introduce class APPLICATION_WINDOW in the gui cluster with three features display_temperature, display_humidity, and display_pressure. APPLICATION_WINDOW is a *subscribed* class: it reacts to the published events by executing the corresponding routine(s), e.g. display_temperature.

[3] This definition complies with the constraint EVENT_DATA -> TUPLE since TUPLE [INTEGER] conforms to TUPLE [4].

Secondly, we introduce class MAIN_WINDOW, which is in charge of subscribing the three features of class APPLICATION_WINDOW listed above to the corresponding three event types (temperature_event, humidity_event, and pressure_event). In order to subscribe feature display_temperature of application_window_1 to event type temperature_event, the subscriber makes the following call:

```
Sensor.temperature_event.subscribe
 (agent application_window_1.display_temperature (?))
```

As a result, feature display_temperature of application_window_1 will be called each time temperature_event is published. The actual argument of feature subscribe in class EVENT_TYPE is an agent expression[4]: **agent** application_window_1.display_temperature(?). The question mark is an open argument that will be filled with concrete event data (value of type INTEGER) when feature display_temperature is executed [2].
Let's have a closer look at feature subscribe of class EVENT_TYPE:

```
subscribe (an_action: PROCEDURE [ANY, EVENT_DATA])
        -- Add an_action to the subscription list.
  require
    an_action_not_void: an_action /= Void
    an_action_not_already_subscribed:
        not has(an_action)
  ensure
    an_action_added:
        count = old count + 1 and has (an_action)
    index_at_same_position: index = old index
```

subscribe takes an argument of type PROCEDURE [ANY, EVENT_DATA].[5] The first formal generic parameter (of type ANY) represents the base type on which the procedure will be called; the second formal generic parameter (of type EVENT_DATA, which is derived from TUPLE), represents the open arguments of the procedure. This procedure will be called when the event is published. It has to be non-void and not already among listed actions, as stated by the precondition of subscribe. This means that the same procedure cannot be subscribed more than once to the same event type. The postcondition ensures that the list of subscribed actions is correctly updated.

2.6 Unsubscribing from an Event Type

Class EVENT_TYPE provides feature *unsubscribe*, which allows objects subscribed to an event type to cancel their subscription. Feature start_actions of class

[4] **agent** x.f(a) is an object representing the operation x.f(a). Such objects, called agents, are used in Eiffel to "wrap" routine calls [2]. One can think of agents as a more sophisticated form of .NET delegates.
[5] This argument is an *agent*.

`MAIN_WINDOW` uses it to unsubscribe `application_window_1` from event type `temperature_event`:

```
Sensor.temperature_event.unsubscribe
  (agent application_window_1.display_temperature)
```

The implementation of `unsubscribe` is similar to that of `subscribe`; it just does the opposite: unsubscribes the procedure from the event type.

2.7 Additional Features of Class EVENT_TYPE

Besides procedures `subscribe`, `unsubscribe`, and `publish` that we have already seen, class `EVENT_TYPE` has three additional features: `suspend_subscription`, `restore_subscription`, and `is_suspended`. It is possible to define custom event types by inheriting from `EVENT_TYPE` and redefining or adding specific features. This is explained in detail in [9].

3 Architecture of the Event Library

In this section, we discuss the architecture of the Event Library. We also show how the library fulfills the requirement of *space*-, *time*-, and *flow-decoupling* of the event-driven paradigm.

3.1 Basic Concepts

The design of the library relies on a few basic concepts: *event type*, *event*, *publisher*, *subscriber*, and *subscribed object*. Let us have a closer look at these notions.

3.1.1 Events and Event Types

The concepts of event and event type are often confused. To reason about event-driven design, in particular in the object-oriented setting, one should understand the difference between them.

In event-driven systems, the interaction between different parts of the application and external actors (such as users, mechanical devices, sensors) is usually based on a data structure called the *event-action table*. This data structure records what action should be executed by some part of the system in response to each *event* caused either by another part of the system, or by an external actor. Thus an *event* is a signal: it represents the occurrence of some action taken by the user (e.g. clicking a button) or a state change of some other parts of the system (e.g. temperature change measured by the sensor). An *event type* provides the abstraction for *events*. In other words, an event is an instance of the corresponding event type. For example, every time the user clicks a mouse button, an event of event type `Mouse_Click` is published. In our sample application, each temperature change caused an event of type `temperature_event` to be published.

3.1.2 Publishers, Subscribers, and Subscribed Objects

Publisher is the part of the system capable of triggering events. The action of triggering an event is called publishing. In the Event Library, this is achieved by calling feature *publish* of the corresponding `EVENT_TYPE` object.

Subscribed objects are notified whenever an event of the corresponding event type is published. The notification is done by calling the feature of the subscribed object previously registered within the event type. In the Event Library, the *agent mechanism* is used for registration and notification of subscribed objects. An object may be subscribed to several event types at a time, and receive notification from all of them. Conversely, an event type may have several subscribed objects.

Subscriber is in charge of registering subscribed objects to a given event type. In the Event Library, this is achieved by calling feature `subscribe` of the corresponding `EVENT_TYPE` object. We introduce a separation between the concepts of subscriber and subscribed object. It is important to note that such distinction provides another level of abstraction in the event-driven design, although in most cases subscribed objects are their own subscribers.

3.2 Implementation

Class `EVENT_TYPE` inherits from the generic class `LINKED_LIST` from the EiffelBase Library [8]. Therefore all features of `LINKED_LIST` are available to the class `EVENT_TYPE`. In a previous version of the Event Library, client-supplier relation was used instead of inheritance. We opted for the inheritance-based solution because it is an easy way to implement the subscription list (class `EVENT_TYPE` can itself be considered as a list of subscribed agents); it also facilitates future extensions of the library, e.g. through the redefinition of features in class `EVENT_TYPE`. On the other hand, it introduces a potential risk: some features inherited from the class `LINKED_LIST` might be misused by the clients of `EVENT_TYPE,` e.g. a client having access to an event type could simply clear the list. Therefore, one may want to hide these features by changing their export status to `{NONE}`, thus preventing the clients from using them directly.

3.3 Space-, flow-, and time-decoupling

Event-driven architectures may provide separation of application layers in three dimensions: *space*, *time*, and *flow*. Such decoupling increases scalability by removing all explicit dependencies between the interacting participants. [5] provides a short survey of traditional interaction paradigms like message passing, RPC, notifications, shared memory spaces, and message queuing; they all fail to provide time, space and flow decoupling at the same time. It is interesting to note that, in spite of its simplicity and small size, the Event Library can provide decoupling in all three dimensions.

Space decoupling. The publisher and the subscribed class do not know each other. There is no relationship (neither client-supplier nor inheritance) between the classes `SENSOR` and `APPLICATION_WINDOW` in Fig. 2. Publishers and subscribed classes are absolutely independent of each other: publishers have no references to the subscribed objects, nor do they know how many subscribed objects participate in the interaction. Conversely, subscribed objects do not have any references to the publishers, nor do they know how many publishers are involved into the interaction. In our sample application, class `SENSOR` has a client-supplier relationship to the class `EVENT_TYPE`, but no relationship to subscribed classes. Only class `MAIN_WINDOW`, which is the *subscriber*, knows the publisher and the subscribed objects. In fact, in the general case, the subscriber does not have to know the publisher; it has to know the event type to which subscribe an action. Event types are declared in the publisher class `SENSOR`; this is why class `MAIN_WINDOW` has a reference to the class `SENSOR`. Had the event types been declared in another class, the subscriber class would keep no reference to the publisher. On the other hand, declaring the event type outside the scope of the publisher (`SENSOR`) might be dangerous: every client having access to the event type (e.g `temperature_event`) can publish new events. In such case, the publisher (`SENSOR`) would have no possibility to control the use of that event type.

Flow decoupling. Publishers should not be blocked while publishing events. Conversely, subscribed objects should be able to get notified about an event while performing other actions; they should not need to "pull" actively for events. Obviously, support for concurrent execution is necessary in order to achieve flow decoupling. In our sample application, there is no such support; therefore flow decoupling does not exist: publisher objects are blocked until all subscribed objects are notified.

The Event Library can ensure flow decoupling, provided that *publisher*, *event type* and *subscribed* objects are handled by independent threads of control, e.g. SCOOP *processors* [7][10]. See section 4 for more details.

Time decoupling. Publishers and subscribed objects do not have to participate actively in the interaction at the same time. Such property is particularly useful in a distributed setting, where publishers and subscribed objects may get disconnected, e.g. due to network problems.

Current implementation of the Event Library does not provide time decoupling. Nevertheless, it can be easily extended to cover this requirement. The basis for such extension should be, as in the case of flow decoupling, a support for concurrent and distributed execution (see section 4).

4 Current Limitations and Future Work

Initially, our goal was to provide a reusable library that implements the *Observer pattern*. We soon realized that the Event Library can be turned into something much more powerful: a simple and easy to use library for event-driven programming. Despite its small size, it caters for most event-based applications. Whenever more advanced features are needed, the library can be easily extended.

An important contribution of our approach is the distinction between the concepts of *subscribed* and *subscriber* objects (see 3.1). Such separation of concepts brings an additional level of abstraction in application design, thus facilitating reasoning about event-driven systems.

Future enhancements of the Event Library could include "conditional event subscription" for subscribed objects only interested in events fulfilling certain criteria or conditions. For example, objects subscribed to event type `temperature_event` may want to be notified of a temperature change only if the value of attribute `temperature` is between 25 and 50 degrees.

A support for concurrent and distributed execution is another important extension of the library. In particular, flow- and time-decoupling cannot be provided without such support (see 3.3). We plan to base the extension on the SCOOP model [7]. SCOOP provides high-level mechanisms for concurrent and distributed object-oriented programming. We are currently implementing the model for Microsoft .NET [10]; other platforms (POSIX threads, MPI) are also targeted.

Acknowledgements

We would like to thank Bernd Schoeller, Susanne Cech, and Vaidas Gasiunas, for their comments. Bertrand Meyer contributed essentially to the design of the library. We also benefited from the very helpful support from Julian Rogers from ESI.

References

1. Arslan V.: Event Library, at http://se.inf.ethz.ch/people/arslan/
2. Eiffel Software Inc.: Agents, iteration and introspection, at http://archive.eiffel.com/doc/manuals/language/agent/agent.pdf
3. Eiffel Software Inc.: EiffelStudio 5.2, at http://www.eiffel.com.
4. Eiffel Software Inc.: Tuples, at http://archive.eiffel.com/doc/manuals/language/tuples/page.html
5. Eugster P. Th., Felber P., Guerraoui R., Kermarrec A.-M.: The Many Faces of Publish/Subscribe, Technical Report 200104 at http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200104.pdf.
6. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edition, Addison-Wesley, 1995.
7. Meyer B.: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997.
8. Meyer B.: Reusable Software: The Base Object-Oriented Component Libraries, Prentice Hall, 1994.
9. Meyer B.: The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design, at http://www.inf.ethz.ch/~meyer/ongoing/events.pdf.
10. Nienaltowski P., Arslan V.: SCOOPLI: a library for concurrent object-oriented programming on .NET, in Proceedings of the 1st International Workshop on C# and .NET Technologies, University of West Bohemia, 5-8 February 2003, Pilsen, Czech Republic.
11. Walden K., Nerson J.-M.: Seamless object-oriented software architecture, Prentice Hall, 1995.