

Event-driven design

Who's in charge?

In the style of programming that we have used so far, the program defines the order of operations. It follows its own scenario, defined by control structures: sequence, conditional, loop. The external world has its say — through user interaction, database access and other input, affecting the conditions that control loops and conditionals; but it's the program that decides when to evaluate these conditions.

In this chapter we explore another scheme, where the program no longer specifies the sequencing of operations directly but is organized instead as a set of *services* ready to be triggered in response to *events*, such as might result from a user clicking a button, a sensor detecting a temperature change, a message arriving on a communication port. At any time, the next event determines which service gets solicited. Once that service has carried out its function, the program gets back to waiting for events.

Such an **event-driven** scheme requires proper initialization: before the real action begins, there must be a setup step to associate services with events.

This architectural style — in the end another control structure, to be added to our previous *catalog* — is also known as **publish-subscribe**, a metaphor emphasizing a division of possible roles between software elements:

← Chapter 7, [Control structures](#).

- Some elements, the *publishers*, may trigger events during execution.
- Some elements, the *subscribers*, register their interest in certain types of events, indicating what services they want provided in response.

These roles are not exclusive, as some subscribers may trigger events of their own. Note that “event” is a *software* concept: even when events originate outside of the software — mouse click, sensor measurement, message arrival — they must be translated into software events for processing; and the software may trigger its own events, unrelated to any external impulse.

Event-driven programming is applicable to many different areas of programming. It has been particularly successful for Graphical User Interfaces (GUI), which we'll use as our primary example.

20.1 EVENT-DRIVEN GUI PROGRAMMING

Good old input

Before we had GUIs, programs would take their input from some sequential medium. For example a program would read a sequence of lines, processing each of them along the way:

```

from
  read_line
  count := 0
until
  exhausted
loop
  count := count + 1
  -- Store last_line at position count in Result:
  Result [last_line] := count
  read_line
end

```



where *read_line* attempts to read the next line of input, leaving it in *last_line*, and *exhausted* doesn't refer to the mood of the programmer but is set to true by *read_line* if there are no more lines to be consumed.

With such a scheme **the program is in control**: it decides when it needs some input. The rest of the world — here a file, or a user typing in lines at a terminal — then has to provide that input.

Modern interfaces

Welcome to the modern world. If you write a program with a GUI, you let users choose, at each step, what they want to do, out of many possibilities — including some unrelated your program, since a user may choose another window, for example to answer an email.

Consider the screen on the adjacent page, from Traffic. The interface that we show to our user includes a text field and a button. There might be many more such “*controls*”. 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 at the top, clicking the button, or any other, such as menu selection.

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

We don't know.

A “control” is a GUI element, such as a window or button. This is Windows terminology; in the Unix world the term is “widgets”.

*A program
GUI*

(Screenshot to
be added.)

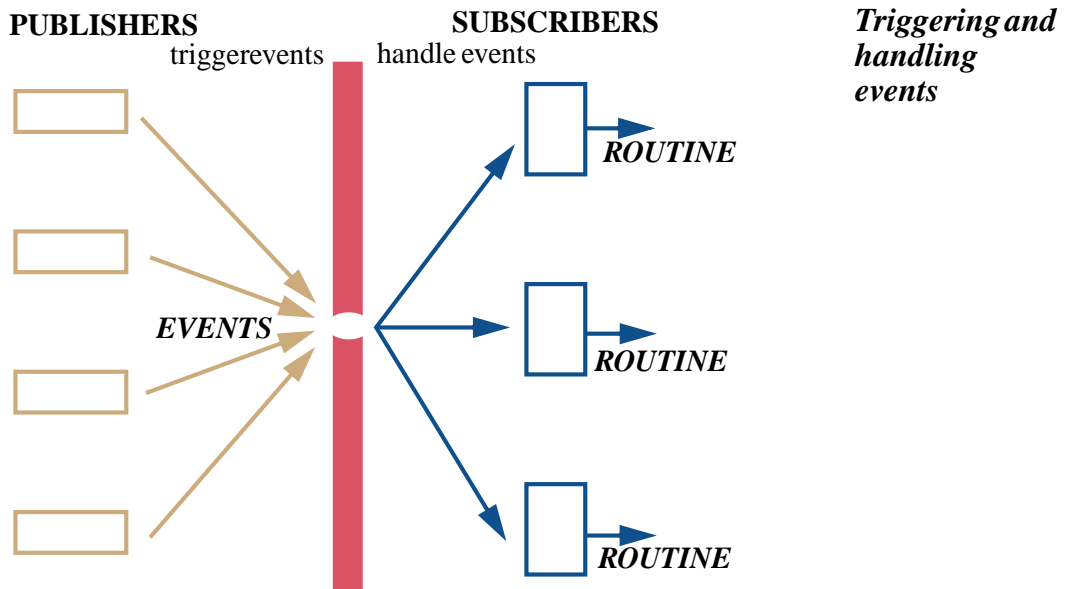
Of course we could use a big **if ... then ... elseif ... end**, or a multi-branch listing all possibilities:

```
inspect  
  user_action  
when "Click on the OK button" then  
  "Save the file"  
when "Input in the File Name field" then  
  "Load a new file"  
when ... Many other branches ...  
end
```

but this suffers from all the problems we have seen with multiple-choice algorithm structures (as part of the justification for dynamic binding): it's big and complex, and highly sensitive to any change in the setup. We want a simpler and more stable architecture, which we won't have to update each time there is a new control.

Event-driven (publish-subscribe) design addresses such a situation through a completely different scheme.

We may picture this scheme as one of those quantum physics experiments (see the figure on the next page) that hurl various particles at some screen pierced with a little hole, to find out what might show up on the other side.



The publish-subscribe style is useful in many different application areas; GUI programming is just an example. You can find many others, in fields such as:

- Communication and networking, where a node on a network may be broadcasting messages that any other node may pick up.
- Process control. This term covers software systems associated with industrial processes, for example in factories. Such a system might have sensors monitoring temperature, pressure, humidity; any new recording, or just those exceeding some preset values, may trigger an event which some elements of the software are prepared to handle.

20.2 TERMINOLOGY

In describing event-driven programming it is important to define the concepts carefully, distinguishing in particular — as in other areas of programming — between types and instances.

Events, publishers and subscribers

Definitions: Event

An **event** is a run-time operation, executed by a software element to make some information (including the information that it occurred) available for potential use by software elements not specified by the operation.

This definition highlights the distinctive properties of events:

- An event releases some **information**. A mouse click should indicate the cursor position; a temperature change, the old and new temperatures.
- Part of the information, always included, is that the event **occurred**: on 5 August 1492, Christopher Columbus set sail; five minutes ago (this is less widely known) I clicked the left button of my mouse. Usually there's more: when and where did Columbus sail? What were the cursor coordinates? But in some cases all that matters is that the event occurred, as with a *timeout* event indicating that a previously set deadline has passed.
- **“Some”** software elements can use this information. This is sufficiently vague to permit various setups: allowing *any* module of the system to find out about events; or ensuring that only certain modules are eligible.
- In all cases, however, what characterizes event-driven design is that **the operation does not name the recipients**. Otherwise an event would just be like a routine call, like $x.f(a, b, c)$, which satisfies all the other properties of the definition: it's an operation that makes information (the arguments a, b, c) available to a software element (the feature f). But when you call a routine you explicitly say whom you are calling. An event is different: it just sends the information out there, for consumption by any software element that has the ability to process it.

Remember that for our purposes an event is a **software** operation; phenomena triggered outside of the software may be called *external events*. An example such as “mouse click event” does not denote the user's click action, an external event, but the result of a GUI library (such as EiffelVision) detecting it and turning it into a software event, for processing by other parts of the software. In addition to such cases, a system may also have its own software-only events.

Some associated terminology, most of it already encountered informally:

Definitions: Trigger, publish, publisher, subscriber

To **trigger** (or **publish**) an event is to execute it. A software element that may trigger events is a **publisher**. A software element that may use the event's information is a **subscriber**.

Remember that an event is defined as operation to be executed.

The same software element may act as both a publisher and a subscriber; in particular it is a common scheme for a subscriber to react to an event by triggering another event.

In the literature you will encounter competitors to the above terms: subscribers are also called **observers**, hence the “Observer pattern” studied next; they are said to **observe** the publishers but also, without fear of mixing sensory metaphors, to **listen** to them, gaining one more name: **listener**. Publishers — the targets of all this visual or auditory attention — are entitled to their own synonym: **subject**.

Arguments and event types

We need a name for the information that comes — according to the definition — with any event:

Definitions: Argument

The information associated with an event (other than the information that the event occurred) constitutes the event's **arguments**.

The term “argument” highlights the similarity with routines. Pushing this similarity further, we'll assume that the arguments are an ordered list, like the arguments in a call $x.f(a, b, c)$. As with routines, the list can be empty; this would be the case in the timeout example.

How do subscribers find out that an event occurred? One model is *polling*: checking repeatedly — as when you subscribe to a newspaper and go see whether the day's edition has been delivered to your mailbox. Another is **notification**: the triggering of an event causes all potential recipients to be notified.

Models for distributing information over the Internet are classified into “pull” (waiting for users to access information) and “push” (sending it to them). The distinction between polling and notification is similar.

The notification model is more flexible and we'll assume it from now on. It can only work if subscribers express their interest in advance, just as you subscribe to a newspaper to receive it every day. But to *what* can you subscribe? It cannot be to an *event*: the event is an operation occurring once: before it's triggered the event does not exist, and afterwards it's too late to subscribe to it! This would be like subscribing to today's newspaper after you've spotted the headline on your neighbor's copy, or retroactively buying shares of a company after the announcement of its latest dividend.

What subscribers need is an **event type**, describing possible events that share general characteristics. For example all left-button mouse clicks are of the same event type, but of a different type from key-press events. This notion of event type plays a central role in event-driven design and will be the central abstraction in our search for a good O-O architecture.

All events of a type have the same *argument type list*. For example, the argument list for any left mouse click event includes the mouse coordinates, two integers. Here too we may borrow a concept from routines, the **signature**, or list of argument types — a procedure *print* (*v*: *VALUE*; *f*: *FORMAT*) has signature [*VALUE*, *FORMAT*], a list of types — and extend it to event types:

Definitions: Event type, signature

Any event belongs to an **event type**.

All events of a given event type have the same argument list **signature**.

For example:

← “[ANATOMY OF A ROUTINE DECLARATION](#)”, page 203. For a function, the signature also includes the result's type.

- A “left click” event type may have signature [*INTEGER, INTEGER*]. It is also possible to have a single “mouse click” event with a third signature element indicating which button was clicked. This is the case in the EiffelVision library, which also adds arguments such as pressure applied, useful (especially in game applications) for joysticks and exotic pointing devices.
- The signature for “temperature change” may be [*REAL, REAL*] to represent old and new temperatures.
- Although we might define an event type for each key on the keyboard, it’s more attractive to use a single “key press” event type of signature [*CHARACTER*], where the argument is the key code.
- For an event type such as “timeout” describing events without arguments the signature is empty, just as with an argument-less routine.

Whenever a publisher triggers an event, it must provide a value for every argument (if any): mouse coordinates, key code, temperatures. This is once again as with routines, where every call must provide actual arguments.

The term “event *type*” may suggest another analogy, where event types correspond to the **types** of O-O programming (classes, possibly with generic parameters), and events to their *instances* (objects). But comparing event types to **routines** is more appropriate; then an *event* of a given type corresponds to one specific *call* to a routine.

In our model, then, an event is not an object — and **an event type is not a class**. Instead the *general notion* of event type is a class, called *EVENT_TYPE* below; and *one particular event type*, for example “left-button mouse click” (the *idea* of left clicks, not that one time last Monday when I distractedly clicked OK to “Delete all?”) is an object. As always when you are hesitating about introducing a class, the criterion is “is this a meaningful data abstraction, with a set of well-understood operations applicable to all instances?”. Here:

- If we decided to build a class to represent a particular event type, its instances would be events of that type; but they have no useful features. True, each event has its own data (the arguments), but there’s no meaningful operation on the event other than accessing such data.
- In contrast, if we treat an event type as an object, the associated features are obvious and useful: trigger a particular event of this type now, with given arguments; subscribe a given subscriber to this event type; unsubscribe a subscriber; list the subscribers; find out how many events of this type have been triggered so far; and so on. This is the kind of rich feature set that characterizes a legitimate class.

Not treating each event as an object is also good for performance, since it is common for execution to trigger many events; each minute move of the cursor is an event, so we should avoid creating all the corresponding objects — even though this does not get us out of the wood since the *arguments* of each event must still be recorded, each represented by a tuple. A good GUI library will remove the performance overhead by recognizing a sequence of contiguous moves in close succession and allocating just one tuple instead of dozens or hundreds.

It is useful to have terms for subscribers’ actions with event types and events:

← Class *EVENT_TYPE*
in the final design:
[“USING AGENTS:
THE EVENT
LIBRARY”](#), 20.5, page
[531](#).

Definitions: Subscribe, register, handle, catch

A software element may become a subscriber to a certain event type by **subscribing** (or **registering**) to it. This is a request to be notified of future events of that event type, so that it can obtain the associated information (arguments) and execute specified actions in response.

When a subscriber gets notified of an event to whose type it has subscribed, it **handles** (or **catches**) the event by executing the registered action.

Although registration (and deregistration) may occur at any time, it is common to have an initialization phase that puts subscriptions in place, followed by the real action, where publishers trigger events which subscribers will handle.

Registering, for a subscriber, means specifying a certain action for execution in response to any event of the specified type. There must be a way for the action to obtain the values of the event's arguments. The obvious way to achieve such registration is to specify a **routine**, whose signature matches the event type's signature. Then an event of the given type will cause a call to the routine, with the event's arguments serving as actual arguments to the call.

We now have the full picture of how an event-driven design works:

- 1 • Some elements, *publishers*, make known to the rest of the system what *event types* they may trigger.
- 2 • Some elements, *subscribers*, are interested in *handling* events of certain event types. They *register* the corresponding actions.
- 3 • At any time, a publisher can *trigger* an event. This will cause execution of actions registered by subscribers for the event's type. These actions will can use the event's arguments.

In the GUI example:

- 1 • A publisher is some element of the software that tracks input devices and triggers events under specified circumstances, for example mouse click or key press. You usually don't have to write such software; rather, you rely on a **GUI library** — EiffelVision for Eiffel, Swing for Java, Windows Forms for .NET... — that takes care of triggering the right events.
- 2 • A subscriber is any element that needs to handle such GUI events; it registers the routines it wants to execute in response For example you may register, for the mouse click event type on a button that says "OK" in a file saving dialog, a routine that saves the file.
- 3 • If, during execution, a user clicks the OK button, this will cause execution of the routine — or routines — registered for the event type.

An important property of this scheme, illustrated by the separation between the two sides of our earlier [figure](#), is that subscribers and publishers do not need to know about each other. More precisely, the definition of "event" *requires* that subscribers do not know the subscribers; the other way around it's more a matter of methodology, and we will see how various architectural solutions fare against this criterion.

← "[Triggering and handling events](#)", [page 512](#).

Keeping the distinction clear

You might think the distinction between events and event types obvious, but in fact — this is a warning, to help you understand the literature if you start using various event-driven programming mechanisms — many descriptions confuse the two; this can make simple things sound tricky.

The following excerpt comes from the introductory presentation of event handling in the online [documentation](#) of .NET, a Microsoft framework whose concepts are reflected in the C# and Visual Basic .NET languages:

From msdn2.microsoft.com/en-us/library/awbf1dfh.aspx, as of August 2006. Numbers, italics and colors added.

Events Overview

Events have the following properties:

- 1 • The publisher determines when an **event** is raised; the subscribers determine what action is taken in response to the **event** .
- 2 • An **event** can have multiple subscribers. A subscriber can handle multiple **events** from multiple publishers.
- 3 • **Events** that have no subscribers are never called.
- 4 • **Events** are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- 5 • When an **event** has multiple subscribers, the *event* handlers are invoked synchronously when an **event** is raised. To invoke **events** asynchronously, see [another section].
- 6 • **Events** can be used to synchronize threads.
- 7 • In the .NET Framework class library, **events** are based on the **EventHandler** delegate and the **EventArgs** base class.

I have highlighted in green those occurrences of “event” where I think the authors really mean event, and in yellow those for which they mean event type (a term that does occur in the .NET documentation, but rarely). Where the word is in *italics*, it covers both. This is all my interpretation, but I think that you will agree. In particular:

- It is not possible (points [1](#), [5](#)) to subscribe to an event; as we’ve seen, the event does not exist until it has been raised, and when it has been raised that’s too late. (Nice idea, though: wouldn’t you like to subscribe retroactively to the event “IBM’s shares rise by at least 5%”?) A subscriber subscribes to an event *type* — to declare that it wishes to be notified of any *event* of that type raised during execution.
- Point [7](#) talks about properties of *classes* describing event *types*, as indeed in .NET every event type must be declared as a class. Such a class must inherit from the “delegate” class **EventHandler** (.NET delegate classes provide a kind of agent mechanism) and use another class **EventArgs** describing the notion of event arguments.

- Point [3](#) sounds mysterious until you realize that it means: “If an *event type* has no subscriber, triggering an *event* of that type has no effect.” So all it describes is an internal optimization: by detecting that an event type has no subscriber, the event mechanism can remove the overhead of raising the corresponding events, which in .NET implies creating an object for each. (The mystery is compounded by the use of “call” for what the rest of the documentation refers to as “raising” an event.)

The possibility of confusion is particularly vivid in two places:

- “A *subscriber can handle multiple events from multiple publishers*” (point [2](#)): this might seem to suggest some sophisticated concurrent computation scheme, where a subscriber catches events from various places at once, but in reality is just a mundane observation: a given subscriber may register for several event types, and several publishers may trigger events of a given type.
- Point [5](#) states that when “*an event*” has multiple subscribers, each will handle it synchronously (meaning right away, blocking further processing) when “*an event*” is raised. Read literally, this would suggest that two “events” are involved! That’s not the idea: the sentence is simply trying to say that when multiple subscribers have registered for a certain event *type*, they handle the corresponding *events* synchronously. It uses a single word, in the same breath, with two different meanings.

So when you read about event-driven schemes remember to ask yourself whether people are talking about events or event types — and (since this is the time for one of our periodic exhortations) please make sure that your own technical documentation defines and uses precise terminology.

Contexts

A subscriber that registers says: “for events of *this type*, execute *that action*”. In practice it may be useful, especially for GUI applications, to provide one more piece of information: “for events of this type occurring *in that context*, execute that action”. For example:

- “If the user clicks the left button *on the OK button*, save the file”.
- “If the mouse enters *this window*, change the border color to red”.
- “If this sensor reports a temperature *above 25° C*, ring the alarm”.

In the first case the “context” is an icon 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.

For GUI programming, a context is usually just a user interface element. As the last example indicates, the notion is more general; a context can in fact be any boolean-valued condition. This covers the GUI example as a special case, taking as boolean condition a property such as “the cursor is on this button” or “the cursor has entered that window”. Here is a general definition:

Definition: Context

In event-driven design, a **context** is a boolean expression specified by a subscriber at *registration* time, but evaluated at *triggering* time, such that the registered action will only be executed if the evaluation yields **True**.

Even though that wasn't event-driven programming, we had a taste of the notion of context when encountering iterators such as *do_if* which performs an action on all the items of a structure that satisfy a certain condition; this is similar to how a context enables a subscriber to state that it is interested in events of a certain type but only if a certain condition holds at triggering time.

← “*The anatomy of an iterator*”, page 477.

We could do without the notion of context by including the associated condition in the registered action itself, which we could write, for example

```
if “The cursor is on the Exit icon” then
    “Normal code for the action”
end
```

but it is more convenient to separate the condition by specifying it, along with the event type and the action, at the time of registration.

20.3 PUBLISH-SUBSCRIBE REQUIREMENTS

With the concepts in place, we will now look for a general solution to the problem of devising an event-driven architecture. We start with the constraints that any good solution must satisfy.

Publishers and subscribers

In devising a software architecture supporting the publish-subscribe paradigm we should consider the following requirements.

- *Publishers must not need to know who the subscribers are:* they trigger events, but, per the basic definition of events, do not know who may process them. This is typically the case if the publisher is a GUI library: the routines of the library know how to detect a user event such as a click, but 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 the compilation, run the payroll, shut down the factory or launch the rocket. To the GUI library, a click is just a click.
- *Any event triggered by one publisher may be consumed by several subscribers.* A temperature change in a factory control system may have to be reflected in many different places that “observe” the event type, for example an alphanumeric temperature display, a graphical display, a database that records all value changes, and a security system that triggers certain actions if the value is beyond preset bounds.
- *The subscribers should not need to know about the publishers.* This is a more advanced requirement, but often desirable too: subscribers know about event types to which they subscribe, but do not have to know where they come from. Remember that one of the aims of event-driven design is to provide a flexible architecture where we can plug in various publishers and various subscribers, possibly written by different people at different times.
- *You may wish to let subscribers register and deregister while the application is running.* The usual scheme is that registration occurs during initialization, to set things up before “real” execution starts; but this is not an obligation, and the extra flexibility may be useful.
- *It should be possible to make events dependent or not on a context.* We have seen the usefulness of binding events to contexts, but the solution should also provide the ability — without having to define an artificial context — just to subscribe to an event regardless of where it happens.
- *It should be possible to connect publishers and subscribers with minimal work.* The actions to be subscribed often come from an existing application, to which you want to add an event-driven scheme. To connect the two sides you’ll have to add some program text, often called “**glue code**”; the less of it the better.

The last requirement is critical to the quality of a system’s architecture, especially when the goal is to build user interfaces: you shouldn’t have to design the core of an application differently because of a particular interface. This observation directly leads to our next notions, model and view.

The model and the view

For user interface design we need not only to separate subscribers from publishers but also to distinguish two complementary aspects of an application:

Definitions: model, view of a software system

The **model** (also called *business model*) is the part of a software system that handles data representing information from the application domain.

A **view** is a presentation of part of that information, in the system's interaction with the outside: human users, material devices, other software.

← [“Definitions: Data, information”, page 10.](#)

“*Application domain*” as used in this definition is also a common phrase, denoting the technical area in which or for which the software operates. For a payroll processing program the application domain is human resources of companies; for a text preparation program it is text processing; for flight control software the application domain is air traffic control.

While the application domain need not have anything to do with software, the “*model*” is a part of the software: the part that deals with that application domain. For payroll processing it is the part of the software that processes information on employees and hours worked, computes salaries, updates the database. For the flight system it's the part that determines airplane itineraries, takeoff times, authorizations and so on. One could say that the model is the part of the software that does the “real job” at hand, independently of interaction with users of the software and the rest of the world.

“Business model” is more precise but we usually just say “model” because the word “business” might be misinterpreted as restricting us to business-oriented application domains (company management, finance etc.) at the expense of engineering domains such as text processing and flight control.

A “*view*” is a presentation of the information, typically for input or output. A GUI is a view: for example a flight system has a user interface allowing controllers to follow plane trajectories and enter commands.

Usually a program covers just one — possibly broad — application domain, but it may have more than one view, hence “*the model*” and “*a view*” in the above definition. It is then good practice to assign the two aspects to two different parts of a system's architecture. In a naïve design for a small program you might not pay much attention to this issue. But in a significant system you should, if only because you may need to plan for *several views*, such as:

- A GUI view (occasionally, several).
- A Web view (“WUI”), allowing use through a Web browser.
- A purely textual (non-graphical) interface, for situations in which graphics support is not available.

- A “*batch*” interface where the system takes its input from a prepared scenario and produces its output globally. This is particularly useful for testing interactive systems: interactive testing is hard as it requires people spending long sessions with the system to try many different combinations. Instead you may prepare a collection of scenarios (typically recorded from sessions with human users) and run them without interaction.
- Views provided by other programs, running locally and accessing the functionality through an API.
- *Web service* views provided by programs running on other computers and accessing the functionality through a Web-directed API. (Web services require specific techniques, such as the SOAP protocol.)

Often one view is enough at the beginning; that’s why it is a common design mistake to build a system with the model and the view intricately connected. Then when you need to introduce other views you may be forced to perform extensive redesign. To avoid this you should practice model-view separation as a general principle, right from the start of a design:

Touch of Methodology: **Model-View Separation Principle**

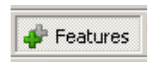
In designing the architecture of a software system, keep the coupling between model elements and view elements to a minimum.

If we use an event-driven model this rule goes well with a clear separation of publishers and subscribers. Both the subscribers and the publishers will interact with the view, but in a decoupled way:

- *Publishers* trigger events which may immediately update the view, typically in minor ways; for example the cursor may change shape when it enters a certain window, and a button usually changes its aspect when it has been pressed (like the Class button on the right).
- *Subscribers* catch events (of event types to which they are subscribed), and process them. The processing may update the view.



Not pressed



Pressed

Note that the publisher-subscriber and model-view divisions are orthogonal: both publishers and subscribers may need to interact with the model as well as with the views, as we can see in the example of a text processing system:

- The need for a publisher to trigger an event may be due to something that happens in a view — a user moves the mouse or clicks a button — or in the model, as when the spell checker detects a misspelled word and a view highlights it.

The wierd seive I recieved
supercedes it's predocessors.

Flagging spelling errors

- The processing of an event by a subscriber will often cause modifications both to the model and to the view. For example if the user has selected a certain text and then presses the Delete key, the effect must be both to remove the selected part from the representation of the text kept internally by the system (model) and to update the display so that it no longer shows that part (view).

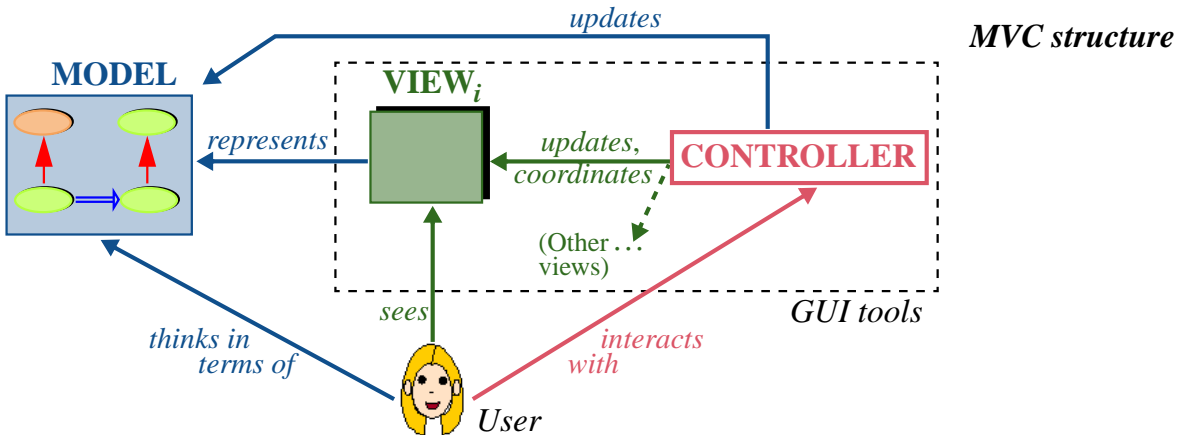
The ~~the~~ next word

Selecting text for deletion

Model-View-Controller

A particularly interesting scheme for GUI design is “Model-View-Controller” or MVC. The role of the third element, Controller, is to direct the execution of an interactive session; this may include creating and coordinating views.

Each of the three parts communicates with the other two:



The presence of a controller provides further separation between the model and the views. (Remember that there may be more than one view, hence “VIEW_{*i*}” in the figure.) The controller handles user actions, which may lead to updates of the view, the model, or both.

As before, a view provides a visual representation of the model or part of it.

The system designer may assume that *users understand the model*: using a text processing system, I should know about fonts, sections and paragraphs; playing a video game, I should have a feel for rockets and spaceships. A good system enables its users to *think* in terms of the model: even though what I see on the screen is no more than a few pixels making up some circular shape, I think of it as a flying vessel. The controller enables me to act on these views, for example by rolling my mouse wheel to make the vessel fly faster; it will then update both the model, by calling features of the corresponding objects to change their attributes (speed, position), and the view, by reflecting the effect of these changes in the visual representation.

The MVC paradigm has had a considerable influence on the spread of graphical interactive applications over the past decades. We will see at the end of this chapter that by taking the notion of event-driven design to its full consequences we can get the benefits of MVC but with a simpler architecture, bypassing some of the relations that populate the last figure.

A side comment on the figure, serving as general advice. Too often in presentations of software concepts you will find impressive diagrams with boxes connected by arrows but little specification of what they mean (their “semantics”). The last figure uses labels such as “*represents*” and “*updates*” to make the semantics clear. (Unlabeled arrows reflect standard conventions for client and inheritance links.) A picture is *not* worth any number of words if it’s just splashes of color. Don’t succumb to the lure of senseless graphics; assign precise semantics to each symbol you use, and document it.

20.4 THE OBSERVER PATTERN

Before we review what will be the definitive scheme for event-driven design (at least for the kind of examples discussed in this chapter), let’s explore a well-known *design pattern*, “Observer”, which also addresses the problem.

About design patterns

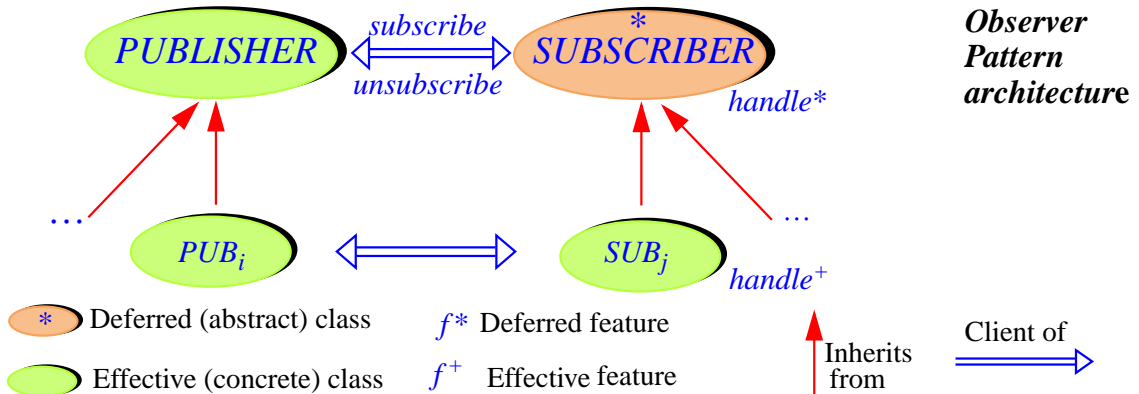
A design pattern is a standardized architecture addressing a certain class of problems. Such an architecture is defined by typical classes that must be part of the solution, their role, their relations — who inherits from whom, who is a client of whom — and instructions for customizing them as the problem varies. Design patterns emerged in the mid-nineties as a way to record and catalog design solutions that good programmers had devised over the years, often reinventing them independently: “*best practices*”.

A couple dozen of these patterns, Observer among them, are widely documented and taught; hundreds more have been described or proposed.

Observer basics

As a general solution for event-driven design, Observer is actually not very good; we’ll analyze its limitations. But you should know about it anyway for several reasons: it’s a kind of classic; it elegantly takes advantage of O-O techniques such as polymorphism and dynamic binding; it may be the best you can do in a language that doesn’t support such notions as agents, genericity and tuples; and it provides a good basis for moving on to the more reasonable solution studied next.

The following figure illustrates a typical Observer architecture. *PUBLISHER* and *SUBSCRIBER* are two general-purpose classes, not specifically dependent on your application; *PUB_i* and *SUB_j* stand for typical publisher and subscriber classes in your application.



**Observer
Pattern
architecture**

Although both *PUBLISHER* and *SUBSCRIBER* are intended to serve as ancestors to classes doing the actual job of publishing and handling events, only *SUBSCRIBER* need be deferred; its deferred procedure *handle* will define, as effected in each concrete subscriber class *SUB_j*, how subscribers handle events. *PUBLISHER* needs no such deferred feature.

We could still make PUBLISHER deferred to prohibit direct instantiation..

As noted, we may say that the subscribers “observe” the publishers, standing on alert for any messages from them (hence the name of the pattern), and that the publishers are the “subjects” of this observation. You will similarly encounter, in the pattern literature, other names for the key features: “attach” for *subscribe*, “detach” for *unsubscribe*, “notify” for *publish*, “update” for *handle*.

The publisher side

Class *PUBLISHER* describes the properties of a typical publisher in charge of an event type — meaning that it can trigger events of that type, through procedure *publish*. The main data structure is a list of *subscribers* to the event type.

```

note
  what: ["Objects that can publish events, all of the same type,
         monitored by subscribers"]
class
  PUBLISHER
feature {SUBSCRIBER} -- Status report
  subscribed (s: SUBSCRIBER): BOOLEAN
    -- Is s subscribed to this publisher?
  do
    Result := subscribers.has (s)
  ensure
    present: has (s)
  end
    
```

```

feature {SUBSCRIBER} -- Element change
  subscribe (s: SUBSCRIBER)
    -- Make s a subscriber of this publisher.
  do
    subscribers.extend (s)
  ensure
    present: subscribed (s)
  end unsubscribe (s: SUBSCRIBER)
    -- Make s a subscriber of this publisher.
  do
    subscribers.remove_all_occurrences (s)
  ensure
    absent: not subscribed (s)
  end
  publish (args: LIST [ANY]) -- Argument Scheme 1
    -- Publish event to subscribers.
  do
    ... See below ...
  end
feature {NONE} -- Implementation
  subscribers: LINKED_LIST [SUBSCRIBER]
    -- Subscribers subscribed to this publisher's event.
end

```

See next about *publish*, the type of its argument, and its “Argument Scheme”.

Procedure *publish* will notify all subscribers that an event (of the event type for which the publisher is responsible) has occurred. It will be easier to write it after devising the class representing a typical subscriber.

The implementation allows calling *subscribe* twice for the same subscriber; then (see *publish* below) the subscriber will execute the subscribed action twice for each event — most likely not the desired effect. To avoid this we could wrap the body of *subscribe* in **if not *subscribed* (*s*) then ... end**, but then a linked list is no longer efficient since *has* requires a traversal. While not critical to the present discussion, this matter must be addressed for any actual use of the pattern; it is the subject of an *exercise*.

→ “*Efficient Observer*”, 20-E.2, page 543.

Apart from *subscribers*, meant for internal purposes only and hence secret (exported to *NONE*), the features are relevant to subscriber objects but not to any others; they are hence exported to *SUBSCRIBER* and so (as you will remember) to the descendants of this class, which indeed need to *subscribe* and *unsubscribe* the corresponding objects. As a general rule, it is a good idea to export features selectively when they are only intended for specific classes and their descendants. Better err on the side of restrictiveness to avoid mistakes caused by classes calling features that are none of their business; it’s easy to ease the restrictions later if you find that new classes need the features.

The subscriber side

```

note
  what: "Object that can register to handle events of a given type"
deferred class
  SUBSCRIBER
feature -- Element change
  subscribe (p: PUBLISHER)
    -- Subscribe to p.
    do
      p.subscribe (Current)
    ensure
      present: p.subscribed (Current)
    end

  unsubscribe (p: PUBLISHER)
    -- Ensure that this subscriber is not subscribed to p.
    do
      p.unsubscribe (Current)
    ensure
      absent: not p.subscribed (Current)
    end

feature {NONE} -- Basic operations
  handle (args: LIST [ANY]) -- Argument Scheme 1
    -- React to publication of one event of subscribed type
    deferred
    end
end

```

See below about the
"Argument Scheme"
and the type of args.

This class is deferred: any application class can, if its instances may need to act as subscribers, inherit from *SUBSCRIBER*. We'll call such descendants "subscriber classes" and their instances "subscribers".

To subscribe to an event type, through the corresponding publisher *p*, a subscriber executes *subscribe* (*p*). Note how this procedure (and, similarly, *unsubscribe*) uses the corresponding feature from *PUBLISHER* to subscribe the current object. That was one of the reasons for exporting the *PUBLISHER* features selectively: it would be useless for a subscriber class to use *subscribe* from *PUBLISHER* directly, since subscribing only makes sense if you provide the corresponding *handle* mechanism; the feature of general interest is the one from *SUBSCRIBER*. (This also justifies using the same names for the features in the two classes, which keeps the terminology simple and causes no confusion since only the *SUBSCRIBER* features are widely exported.)

Procedure *unsubscribe* removes an observer from the attention of the corresponding publisher. To avoid *memory leaks*, do not forget to call it when a subscriber no longer needs its subscription. This recommendation also applies to other architectural techniques and is further discussed [below](#).

→ "[SUBSCRIBER DISCIPLINE](#)", 20.6,
[page 536](#).

Each observer class will provide its own version of *handle*, describing how it handles an event. The not so pleasant part is accessing arguments if any; that's because we tried to make *PUBLISHER* and *SUBSCRIBER* general, and so had to declare *args*, representing the event arguments in both *publish* and *handle* in these respective classes, a completely general type, *LIST [ANY]*; but then *handle* has to force the right type and number of arguments if it can. For example, to process a mouse click event with the $[x, y]$ coordinates as argument — by calling some *operation* which takes these values as its own routine arguments — we may use

```

handle (args: LIST [ANY])                                -- Argument Scheme 1
  -- React to publication of mouse click event by performing
  -- operation on the cursor coordinates.
do
  if args.count >= 2 and then
    ({x: REAL} (args.item (1)) and {y: REAL} (args.item (2)))
  then
    operation (x, y)
  else
    -- Do nothing, or report error
  end
end

```

The Object Tests make sure that the first and second elements of the *args* list are *REALs*, and bind them to *x* and *y* within the **then** clause. The only way to avoid this awkward run-time testing of argument types would be to specialize *PUBLISHER* and *SUBSCRIBER* by declaring the exact arguments to *publish* and *subscribe*, for example

```

publish (x, y: REAL)                                -- Argument Scheme 2

```

and similarly for *handle* in *SUBSCRIBER*. This loses the generality of the scheme since you can't use the same *PUBLISHER* and *SUBSCRIBER* classes for event types of different signatures. Although it's partly a matter of taste, I would actually recommend this "**Argument Scheme 2**" if you need to use the Observer pattern, because it will detect type errors — a publisher passing the wrong types of arguments to an event — at compile time, where they belong.

With *handle* as written above you'll only find them at run time, through the tests on the size and element types of *args*; that's too late to do anything serious about the issue, as reflected by the rather lame "**Do nothing, or report error**" above: doing nothing means ignoring an event (is that what we want, even if the event is somehow deficient since it doesn't provide the right arguments?); and if we report an error, report it to whom? The message should be for the developers — us! — but it's the poor end user who will get it.

It was noted in the discussion of object test that this mechanism should generally be reserved for objects coming from the outside, not those under the

← "**ENFORCING A TYPE: OBJECTTEST**",
18.2, page 464.

program’s direct control, for which the designer is in charge of guaranteeing the right types statically. Here the publishing and handling of arguments belong to the same program; using object test just doesn’t sound right.

It is actually possible to obtain a type-safe solution by making classes *PUBLISHER* and *SUBSCRIBER* generic; the generic parameter is a tuple type representing the signature of the event type (that is to say, the sequence of argument types). That solution will appear in the final publish-subscribe architecture below (“Event Library”). We won’t develop it further for the Observer pattern because it relies on mechanisms — tuple types, constrained genericity — that are not all available in other languages: if you are programming in Eiffel, which has them, you should use that final architecture (relying on agents), which is better than an Observer pattern anyway and is available through a prebuilt library. It is a good exercise, however, to see how to improve Observer through these ideas; try it now on the basis of the hints just given, or wait until you have seen the solution below.

→ “[Type-safe Observer](#)”, 20-E.3, page 543.

Publishing an event

The only missing part of the Observer pattern’s implementation is the body of the *publish* procedure in *PUBLISHER*, although I hope you have already composed it in your mind. It’s where the pattern gets really elegant:

```
publish (args: ... Argument Scheme 1 or 2, see above discussion ...)
  -- Publish event to subscribers.
  do
    -- Ask every subscriber in turn to handle the message:
    from subscribers.start until subscribers.after loop
      subscribers.item.handle (args)
    subscribers.forth
  end
end
```

← To be inserted in class *PUBLISHER*, page 526.

(With “[Argument Scheme 2](#)” the arguments passed to *handle* will be more specific, for example *x* and *y* for a mouse click.) The highlighted instruction takes advantage of polymorphism and dynamic binding: *subscribers* is a polymorphic list; each item in the list may be of a different *SUBSCRIBER* type, characterized by a specific version of *handle*; dynamic binding ensures that the right version is called in each case.

← Page 528.

Assessing the Observer pattern

The Observer pattern is widely known and used; it is an interesting application of object-oriented techniques. As a general solution to the publish-subscribe problem it suffers from a number of limitations:

- The argument business, as discussed, is unpleasant, causing a dilemma between two equally unattractive schemes: awkward, type-unsafe run-time testing of arguments, and specific, quasi-identical *PUBLISHER* and *SUBSCRIBER* classes for each event type signature.
- Subscribers directly subscribe to publishers. This causes undesirable coupling between the two sides. Subscribers shouldn't need to know which part of an application or library triggers certain events. What we are really missing here is an intermediary — a kind of broker — between the two sides. The more fundamental reason is that the design misses a key abstraction: the notion of event type, which it merges with the notion of publisher.
- With a single general-purpose *PUBLISHER* class, a subscriber may register with only one publisher; with that publisher, it can register only one action, as represented by *handle*; as a consequence it may subscribe to only one type of event. This is severely restrictive. An application component should be able to register various operations with various publishers. It is possible to address this problem by adding to *publish* and *handle* an argument representing the publisher, letting subscribers discriminate between publishers; this solution is detrimental to modular design since the handling procedures will now need to know about all events of interest. Another technique is to have several independent publisher classes, one for each type of event; this addresses the issue but sacrifices reusability.
- Because publisher and subscriber classes must inherit from *PUBLISHER* and *SUBSCRIBER*, it is not easy to connect an *existing* model to a new view, for example a GUI, without adding significant glue code. In particular, you can't directly reuse an existing procedure from the model as the action to be registered by a subscriber: you have to fill in the implementation of *handle* so that it will call that procedure, with the arguments passed by the publisher.
- The last problem gets worse in languages without multiple inheritance. *PUBLISHER* and *SUBSCRIBER*, intended to be inherited by publisher and subscriber classes, both need effective features: respectively *publish*, with its fundamental algorithm, and *subscribe*. In Java, C# and other languages that do not support multiple inheritance from classes with effective features, this prevents publisher and subscriber classes from having other parents as may be required by their role in the model. The only solution is to write special publishers and suppliers — more glue code — and make them clients of the corresponding model classes .

- Note finally that the classes given above already correct some problems that arise with standard implementations of the Observer pattern in the literature. For example the usual presentation binds a subscriber to a publisher at *creation* time, using the publisher as an argument to the observer’s creation procedure. The above implementation provides instead a *subscribe* procedure in *OBSERVER*, to bind the observer to a specific publisher when desired; so at least you can later unsubscribe, and re-subscribe to a different publisher.

All these problems have not prevented designers from using Observer successfully, but they have two serious consequences. First, the resulting solutions lack flexibility; they may cause unnecessary work, for example writing of glue code, and unnecessary coupling between elements of the software, which is always bad for the long-term evolution of the system. Second, they are not *reusable*: each programmer must rebuild the pattern for every system that needs it, adapting it to the system’s particular needs.

The preceding assessment of “Observer” is an example of how one may analyze a proposed *software architecture*. Use it as a guide when presented with possible design alternatives. The criteria are always the same: reliability (decreasing the likelihood of bugs), reusability (minimizing the amount of work to integrate the solution into a new program), extendibility (minimizing adaptation effort when the problem varies), and simplicity.

→ See “[Touch of Methodology: Assessing software architectures](#)”, page 541.

20.5 USING AGENTS: THE EVENT LIBRARY

We are now going to see how, by giving the notion of event type its full role, we can obtain a solution that removes all these limitations. It is not only more flexible than what we have seen so far, and fully reusable (through a library class that you can use on the sole basis of its API); it’s also much simpler. The key boost comes from the agent and tuple mechanisms.

Basic API

We focus on the essential data abstraction resulting from the discussion at the beginning of this chapter: event type. We won’t have *PUBLISHER* or *SUBSCRIBER* classes any more, but just one class — yes, a single class solves the entire problem — called *EVENT_TYPE*.

Fundamentally, two features characterize an event type:

- **Subscribing**: a subscriber object can register its interest in the event type by subscribing a specified action, to be represented by an agent.
- **Publishing**: triggering an event.

We benefit from language mechanisms to take care of the most delicate problems identified in the last section:

- Each event type has its own signature. We can define the signature as a tuple type, and use it as generic parameter to *EVENT_TYPE*.
- Each subscription should subscribe a specific action. We simply pass this action as an agent. This allows us in particular to reuse an existing feature from the business model.

These observations are enough to give the interface of the class:

```

note
  what: "Event types, allowing publishing and subscribing"
class EVENT_TYPE [ARGUMENTS → TUPLE] feature
  publish (args: ARGUMENTS)
    -- Trigger an event of this type.

  subscribe (action: PROCEDURE [ANY, ARGUMENTS])
    -- Register action to be executed for events of this type.

  unsubscribe (action: PROCEDURE [ANY, ARGUMENTS])
    -- De-register action for events of this type.
end

```

*Class interface only.
The implementations of
publish and subscribe
appear below.*

If you are an application developer who needs to integrate an event-driven scheme in a system, the above interface — for the class as available in the Event Library — is all you need to know. Of course we'll explore the implementation too, as I am sure you'll want to see it. (It will actually be more fun if you try to devise it yourself first.) But for the moment let's look at how a typical client programmer, knowing only the above, will set up an event-driven architecture.

Using event types

The first step is to define an event type. This simply means providing an instance of the above library class, with the appropriate actual generic parameters. For example, you can define

```

left_click: EVENT_TYPE [ TUPLE [x: REAL; y: REAL] ]
  -- Event type representing left-button click events
once
  create Result
end

```

[1]

This defines an object that represents the event type. Remember, we don't need an object per event; that would be a waste of space. We only need an object per event *type*, such as left-click. Because this object must be available to several parts of the software — publishers and subscribers — the execution needs just one instance; that's where **once** come in handy. One of the advantages is that you don't need to worry about when to create the object; whichever part of the execution first uses *left_click* will (unknowingly) do it.

We'll see in just a moment where this declaration of the event type should appear; until then let's assume that subscriber and publisher classes both have access to it.

To trigger an event, a publisher — for example a GUI library element that detects a mouse click — simply calls *publish* on this event type object, with the appropriate argument tuple; in our example:

```
left_click.publish ([your_x, your_y])
```

On the subscriber side things are equally simple; to subscribe an action represented by a procedure *p* (*x*, *y*: *REAL*), it suffices to use

```
left_click.subscribe (agent p) [2]
```

This scheme has considerable flexibility, achieved in part through the answer to the pending question of where to declare the event type:

- If you want to have a single event type published to all potential subscribers, just make it available to both publisher and subscriber classes by putting its declaration [1] in a “facilities” class to which they all have access, for example by inheriting from it.
- Note, however, that the event type is just an ordinary object, and the corresponding features such as *left_click* just ordinary features that may belong to any class. So the publisher classes — for example classes representing graphical widgets, such as *BUTTON*, in a library such as EiffelVision — may declare *left_click* as one of their features. Then the scheme for a typical subscription call becomes

```
your_button.left_click.subscribe (agent p) [3]
```

This allows a subscriber to monitor — “observe” or “listen to”, in Observer pattern terminology — mouse events from one particular button of the GUI. Such a scheme implements the notion of **context** introduced [earlier](#); here the context is the button.

←“Contexts”, page 518.

Whenever the context is relevant — subscribers don’t just subscribe to an event type as in [2], but to events occurring in a context, as in [3] — the proper architectural decision is to declare the relevant event types in the corresponding context classes. The declaration of *left_click* [1] becomes part of a class *BUTTON*. It remains a **once** function, since the event type is common to all buttons of that kind; the event type object will be created on the first *subscribe* or *publish* call (whichever comes first). If left-click is relevant for several kinds of widget — buttons, windows, menu entries ... — then each of the corresponding classes will have an attribute such as *left_click*, of the same type. The once mechanism ensures, as desired, that there is one event type object — more precisely: at most one — for each of these widget types.

This is the technique used by EiffelVision.

So we get the appropriate flexibility, and can tick off the last remaining item (“*It should be possible to make events dependent or not on a context*”) on our list of [requirements](#) for a publish-subscribe architecture:

←“Publishers and subscribers”, page 519.

- For events that are relevant independently of any context information, declare the event type in a generally accessible class.)
- If a context is needed, declare the event type as a feature of a class representing contexts; it will be accessible at run time as a property of a specific context object.

In the first case, the event type will have at most one instance, shared by all subscribers. In the second case, there will be at most one instance for each context type for which the event type is relevant.

Event type implementation

Now for the internal picture. It remains to see the implementation of *EVENT_TYPE*. It is similar to the above implementation of a *PUBLISHER*. A secret feature *subscribers* keeps the list of subscribers. Its signature is now

```
subscribers: LINKED_LIST [PROCEDURE [ANY, ARGUMENTS]]
```

(where, as before, *LINKED_LIST* is a naïve structure but sufficient for this discussion; for a better one look up the actual class text of *EVENT_TYPE* in the Event Library, or do the [exercise](#)). The items we store in the list are no longer “subscribers”, a notion that the architecture no longer requires, but simply agents, with a precise type: they must represent procedures that take arguments of the tuple type *ARGUMENTS*, as defined for the class. This considerably improves the type safety of the solution over what we saw previously: mismatches will be caught at compile time as bad arguments to *subscribe*.

→ “[Efficient Observer](#)”, 20-E.2, page 543.

For *subscribe* it suffices (in the “naïve” implementation) to perform

```
subscribe (action: PROCEDURE [ANY, ARGUMENTS])
  -- Register action to be executed for events of this type.
do
  subscribers.extend (action)
ensure
  present: subscribers.has (action)
end
```

The use of *ARGUMENTS* as the second generic parameter for the *PROCEDURE* type of *action* ensures compile-time rejection of procedures that do not take arguments of a matching type.

To publish an event we traverse the list and call the corresponding agents. This is in fact the same [code](#) as in class *PUBLISHER* for the Observer pattern, ← [Page 529](#). although *args* is now of a more appropriate type, *ARGUMENTS*:

```
publish (args: ARGUMENTS)
  -- Publish event to subscribers.
do
  -- Trigger an event of this type.
  from subscribers.start until subscribers.after loop
    subscribers.item.call (args)
  subscribers.forth
end
end
```

Any argument to the agent feature *call* must be a tuple; this is indeed the case since *ARGUMENTS* is constrained to be a tuple type.

The solution just describes is at the heart of the “Event Library”, and also of the EiffelVision GUI library; it is widely used for graphical applications, some small and some complex, including the EiffelStudio environment itself.

20.6 SUBSCRIBER DISCIPLINE

If you apply any of the techniques of this chapter, from the crude Observer pattern to the agent-based mechanism, you should be aware of a performance issue, leading to potentially disastrous “memory leaks” but easy to avoid if you pay attention to the subscribers’ behavior:

Touch of Methodology:
Do not forget to unsubscribe

If you know that after a certain stage of system execution a certain subscriber will no longer need to be notified of events of a certain event type, do not forget to include the appropriate call to *unsubscribe*.

Why this rule? The problem is memory usage. It is clear from the implementation of *subscribe* — both the version from *PUBLISHER* in the Observer pattern and the version from *EVENT_TYPE* in the Event Library approach — that registering a subscriber causes the publisher to record, in its list *subscribers*, a reference to the subscriber object. In a GUI application the publisher belongs to a view, and the subscriber to the model. So a view object retains a reference to a model object, which itself may directly and indirectly refer to many other model objects (say planes, flights, schedules and so on in our flight control example). But then it becomes impossible — unless the view objects themselves go away — to get rid of any such model object even if the computation does not need it any more. In a modern environment with garbage collection, such as Eiffel, the GC will never be able to reclaim the objects as long as others refer to them. In an environment with manual memory reclamation (C, C++), it’s even worse. In either case we have a source of “memory leak”: as execution fails to return unneeded space, memory occupation continues to grow.

← Page 526.
← Page 532.

Hence the above rule: subscribing is great, but once you no longer need a service do not forget — as with free magazines and catalogs, if you don’t want to see your mailbox inexorably fill up — to unsubscribe.

Methodological rules are never as effective as tools and architectures that guarantee the desired goal. In this case, however, there is no obvious way to enforce unsubscription, other than through this methodological advice.

20.7 SOFTWARE ARCHITECTURE LESSONS

The designs reviewed in this chapter prompt some general observations about software architecture.

Choosing the right abstractions

The most important issue in software design, at least with an object-oriented approach, is to identify the right classes — data abstractions. (The second most important issue is to identify the relations between these classes.)

In the Observer pattern, the key abstractions are “Publisher” and “Subscriber”. Both are useful concepts, but they turn out to yield an imperfect architecture; the basic reason is that these are not good enough abstractions for the publish-subscribe paradigm. At first sight they would appear to be appropriate, if only because they faithfully reflect the two words in the general name for the approach. What characterizes a good data abstraction, however, is not an attractive name but a set of consistent features. The only significant feature of a publisher is that it publishes events from a given event type, and the only significant feature of a subscriber is that it can subscribe to events from a given event type. That’s a bit light.

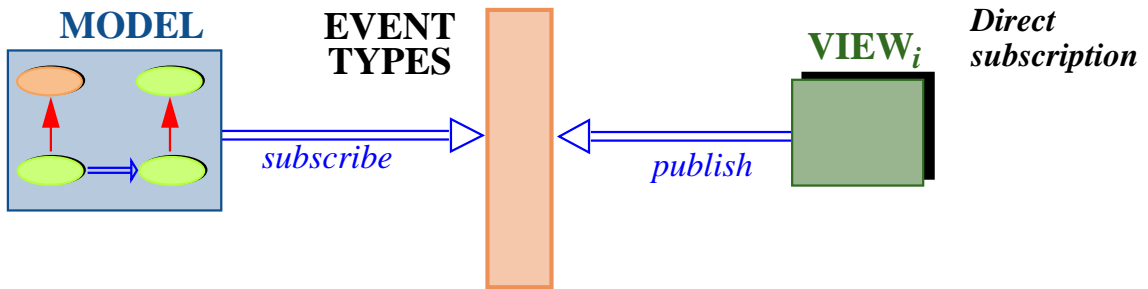
The more significant data abstraction, not recognized by the Observer design, is the notion of event type. This is a clearly recognizable abstraction with several characteristic features: commands to publish and subscribe events, and the notion of argument (which could be given more weight through a setter command and a query). So it meets the criteria.

By treating *EVENT_TYPE* as the key abstraction, yielding the basic class of the final design, we avoid forcing publisher and subscriber classes to inherit from specific parents. A publisher is simply a software element that uses *publish* for some event type, and a subscriber an element that uses *subscribe*.

MVC revisited

One of the consequences of the last design is to simplify the overall architecture suggested by the Model-View-Controller paradigm. The Controller part is “glue code” and it’s good to keep it to the strict minimum.

EVENT_TYPE provides the heart of the controller architecture. In a simple scheme it can actually be sufficient, if we let elements of the model subscribe directly to events:



(The double arrows represent, as usual, the client relation, used here to implement the more abstract relations of the general MVC picture.) In this scheme there is no explicit controller component.

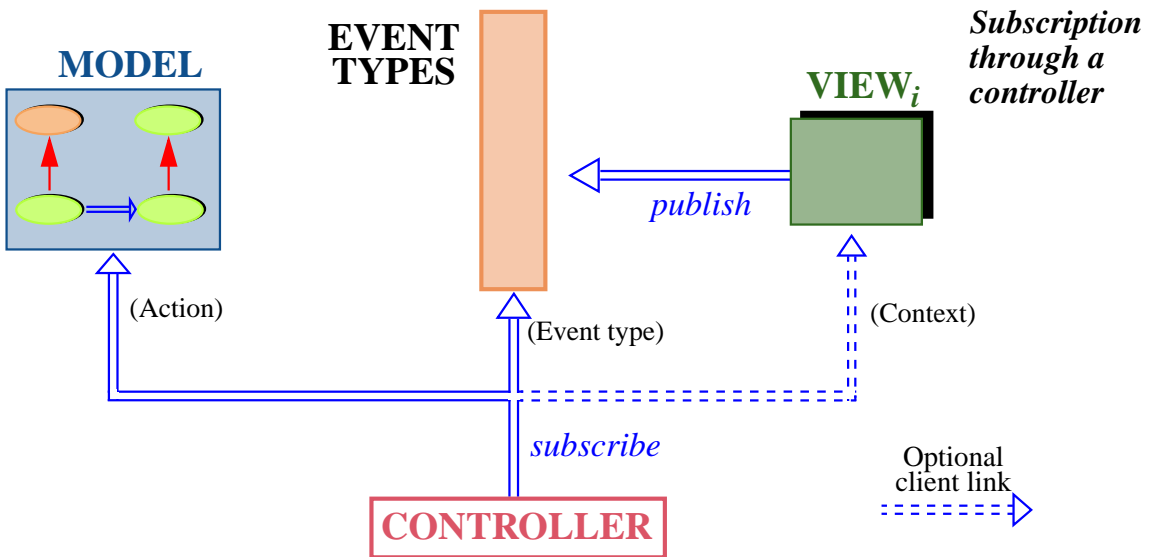
While the model does not directly know about the view (if it does not use contexts), it does connect to specific event types. This setup has both limitations and advantages:

- On the negative side, it can make it harder to change views: while we are not limited to a single view, any new view should trigger the same events. This assumes that the various views are not entirely dissimilar; for example one could be a GUI and another a Web interface.
- It has, on the other hand, the merit of great simplicity. Model elements can directly specify which actions to execute for specific events coming from the interface. There is essentially no glue code.

This scheme is good for relatively simple programs where the interface, or at least the interface style, is known and stable. For more sophisticated cases, we may reintroduce an explicit controller, taking the task of event type subscription away from the model as illustrated on the next figure (top of the adjacent page). The controller is now a separate part of the software. Its job is to subscribe elements of the model to event types; it will have connections both to:

- The model since the arguments to *subscribe* are actions to be subscribed, and these must be agents built from the mechanisms of the model.
- The view, if contexts are used. The figure shows this as an optional client link. ← As per style [3], page 533.

This solution achieves complete uncoupling between model and view; in a typical application the controller will still be still a small component, achieving the goal of using as little glue code as possible.



The model as publisher

In the GUI schemes seen so far all events come from the GUI, normally through the mechanisms of a library, and are processed by the model. In other words the views are publishers, and model elements are subscribers.

It is possible to extend the scheme to let the model publish too. For example if an element of the GUI such as pie chart reflects a set of values which the model may change, the corresponding model elements may publish change events. Views become subscribers to such events.

This possibility is easy to add to the second scheme, subscription through a controller. The controller will now act as a fully bidirectional broker, receiving events from the views for processing by the model and the other way around.

This solution, which adds complexity to the controller, is only useful in the case of multiple views.

Invest then enjoy

Common to the two architectures we have seen, Observer and Event Library, is the need to subscribe to event types prior to processing them.

It is possible for subscribers to subscribe and unsubscribe at any time; in fact, with the Event Library solution, the program can create new event types at any stage of the computation. While this flexibility can be useful, the more typical usage scenario clearly divides execution into two steps:

- During initialization, subscribers register their actions, typically coming from the model.
- Then starts execution proper. At that stage the control structure becomes event-driven: execution proceeds as publishers trigger events, which (possibly depending on the contexts) cause execution of the subscribed model actions.

(So from the order of events it's really the "Subscribe-Publish" paradigm.) Think of the life story of a successful investor: set up everything, then sit back and prosper from the proceeds.

You may remember a variant of the same general approach, the "compilation" strategy that worked so well for topological sort: first translate the data into an appropriate form, then exploit it.

← [*"Interpretation vs compilation", page 454.*](#)

Assessing software architectures

The key to the quality of a software system is in its architecture, which covers such aspects as:

- The choice of classes, based on appropriate data abstractions.
- Deciding which classes will be related, with the overall goal of minimizing the number of such links (to preserve the ability to modify and reuse various parts of the software independently).
- For each such link, deciding between client and inheritance.
- Attaching features to the appropriate classes.
- Equipping classes and features with the proper contracts.
- For these contracts, deciding between a "demanding" style (strong preconditions, making the client responsible for providing appropriate values), a "tolerant" style (the reverse), or an intermediate solution.
- Removing unneeded elements.
- Avoiding code duplication and removing it if already present; techniques involve inheritance (to make two or more classes inherit from an ancestor that captures their commonality) as well as genericity, tuples and agents.
- Taking advantage of known design patterns.
- Devising good APIs: simple, easy to learn and remember, equipped with the proper contracts.
- Ensuring consistency: throughout the system, similar goals should be achieved through similar means. This governs all the aspects listed so far; for example, if you use inheritance for a certain class relationship, you shouldn't use client elsewhere if the conditions are the same. Consistency is also particularly important for an API, to ensure that once programmers have learned to use a certain group of classes they can expect to find similar conventions in others.

Such tasks can be carried out to improve existing designs, an activity known as *refactoring*. It's indeed a good idea always to look at existing software critically, but prevention beats cure. The best time to do design is the first.

Whether as initial design or as refactoring, work on software architecture is challenging and rewarding; the discussion in this chapter — and a few others in this book, such as the development of topological sort — give an idea of what it involves. The criteria for success are always the same:

Touch of Methodology:
Assessing software architectures

When examining possible design solutions for a given problem, discuss alternatives critically. The key criteria, are: reliability, extendibility, reusability, and simplicity.

20.8 FURTHER READING

Trygve Reenskaug, MVC papers at heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

Trygve Reenskaug, a Norwegian computer scientist, introduced the Model-View-Controller pattern while working at Xerox PARC (the famed Palo Alto Research Center) in 1979. The page listed contains a collection of his papers on the topic. I find his original 1979 MVC memo (barely more than a page) still one of the best presentations of MVC.



Reenskaug

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns*, Addison-Wesley, 1994.

A widely used reference on design patterns. Contains the standard description of Observer, along with many others, all expressed in C++.

Bertrand Meyer: *The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design*, in *From Object-Oriented to Formal Methods: Essays in Memory of Ole-Johan Dahl*, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, Lecture Notes in Computer Science 2635, Springer-Verlag, 2004, pages 236-271. Available online at se.ethz.ch/~meyer/publications/lncs/events.pdf.

A significant part of the present chapter's material derives from this article, which analyzes the publish-subscribe pattern in depth, discussing three solutions: Observer pattern, .NET delegate mechanism, and the event library as presented above.

20.9 KEY CONCEPTS LEARNED IN THIS CHAPTER

- Event-driven design, also called “publish-subscribe”, leads to systems whose execution is driven by responses to events rather than by traditional control structures. The events are triggered by the software, often in reaction to external events. GUI programming is one of the important areas of application.
- The key abstraction in event-driven design is the notion of event type.
- *Publishers* are software elements that may trigger events of a certain event type. *Subscribers* are elements that request to be notified of events of a certain type by *registering* actions to be executed in response.
- In a system with one or more interfaces or “views”, an important design guideline is to keep the views separate from the core or the application, known as the “model”.
- The Model-View-Controller architecture interposes a “controller” between the model and the view to handle interactions with users.
- The Observer pattern addresses event-driven design by providing high-level classes *PUBLISHER* and *SUBSCRIBER*, from which publisher and subscriber classes must respectively inherit. Every subscriber class provides an *update* procedure to describe the action to be executed in response to event. Internally, each publisher object keeps a list of its subscribers. To trigger an event, it calls *update* on its subscribers; thanks to dynamic binding, each such call executes the desired version.
- Agents, constrained genericity and tuples allows a general solution to event-driven design through a single reusable class based on the problem’s central abstraction: *EVENT_TYPE*.
- Software architecture is the key to software quality. Devising effective architectures, and improving existing ones (refactoring) should be a constant effort, focused on simplicity and striving at reliability, extendibility and reusability.

New vocabulary

Application domain	Argument (of an event)	Business model
Catching (an event)	Context (of an event)	Control (Windows)
Controller	Event	Event-driven
Event type	External event	Glue code
Handling (an event)	Model	MVC
Publish (an event)	Publish-Subscribe	Register
Refactoring	Signature (of event type)	Subscribe
Trigger (an event)	View	Widget

20-E EXERCISES

20-E.1 Vocabulary

Give a precise definition of each of the terms in the above vocabulary list.

20-E.2 Efficient Observer

Choosing the appropriate representation of the subscribers list, adapt the implementation of the Observer pattern so that the following operations are all **O(1)**: add a subscriber (doing nothing if it was already subscribed); remove a subscriber (doing nothing if it was not subscribed); find out if a potential subscriber is subscribed. The *publish* procedure, ignoring the time taken by subscribers' actual handling of the event, should be **O(count)** where *count* is the number of subscribers actually subscribed to the publisher. Overall space requirement for the *subscribers* data structure should be reasonable, e.g. **O(count)**. (*Hint*: look at the various data structures of chapter 10 and at the corresponding classes in EiffelBase.) Note that this optimization also applies to the Event Library implementation.

← [“THE OBSERVER PATTERN”, 20.4, page 524.](#)

20-E.3 Type-safe Observer

Show that in implementing the Observer pattern a type scheme is possible that removes the drawbacks of both “Argument Scheme 1” and “Argument Scheme 2” by taking advantage, as in the last design of this chapter (Event Library), of tuple types and constrained genericity. Your solution should describe how the *PUBLISHER* and *SUBSCRIBER* classes will change, and also present a typical publisher and subscriber classes inheriting from these.

← [“The subscriber side”, page 527.](#)

