This is a pre-publication version of an article that appeared in the JOOP Eiffel column in 1999. Citation reference: Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer and Emmanuel Stapf, *From Calls to Agents*, in *Journal of Object-Oriented Programming* (JOOP), vol. 12, no 6, June 1999.

From calls to agents

Paul Dubois, Mark Howard, Bertrand Meyer, Michael Schweitzer and Emmanuel Stapf

Agents bring a new level of expressive power by providing elegant mechanisms for iteration, numerical computation, financial applications, introspection, and type-safe higher-level functionals.

Of the recent Eiffel extensions previewed in the last column ("Extension Season", June 1999, see http://www.inf.ethz.ch/personal/meyer/publications/joop/extensions.pdf), some are intended to clean up and simplify existing mechanisms, but a few significantly raise the expressive power of the language. The most spectacular of these is *agents*, a new mechanism for manipulating operations as objects — so important on its own that it deserves a presentation of its own.

Like any good language extension, agents address many needs at once. Among their applications we may note:

- **Iterators**: mechanisms that will apply a routine call to all the elements of a certain structure, such as a list, substituting each of its elements in turn for the target of the call, or one of its arguments.
- **Numerical computation**: passing a numerical function as argument to a mechanism that will apply it to many different values, as in the typical example of an integration routine.
- **Financial applications**: factoring out repetitive operations on objects representing such abstractions as shares or company histories.
- **Offloading computation**: we may pass an agent to another software element, hence giving it the right to operate on some of our own data structures, at a time of its own choosing.
- **Initialization**: using an agent to describe an initial operation to be applied to all future objects of a given type.
- **Introspection**: using agents to gain information about routines and other software elements, as part of Eiffel's introspective capabilities, enabling a software system, during execution, to explore and manipulate information about its own properties. (Introspection is also known as *reflection*.)

The agent mechanism, addressing these needs, is *typd*: agents, representing routines ready to be applied to some arguments, have a precise type and benefit from all the Eiffel mechanisms for type checking.

Another test of a good extension, distinguishing it from mere "featurism", is that it should not confuse users by introducing alternative ways of doing things that were perfectly doable before. Agents satisfy this requirement since the goals just listed did not previously have full language support. Some, such as introspection, could be achieved only through library mechanisms; others, such as iterations, were possible before (also through libraries) but are much more convenient and general now; yet others, such as integration and other numerical applications, required going through the CECIL library, the C-Eiffel Call-In Library, and hence were not type-safe. Agents provide simpler and safer ways to achieve these goals; in areas where existing language mechanisms had proved sufficient, agents do not affect Eiffel practice.

Writing an agent expression

An agent is an object, representing a routine ready to be evaluated. To obtain an agent, you will write an *agent expression*, which will refer to that routine. An agent expression is easy to recognize since it will always include a tilde character ~ (the only use of that character in the language, except of course in strings).

Consider a class *C* with a procedure

p (*x*: *T* ; *y*: *U*)

Then, with *c1* of type *C*, *t1* of type *T*, *u1* of type *U*, a possible agent expression is

agent *c1*. *p* (*t1*, *u1*)

This agent expression superficially resembles a call to the procedure *p*, such as

c1.*p* (*t1*, *u1*)

but is quite different: the agent expression doesn't call the procedure; it simply denotes an object that has, in itself, the ability to call it. So if we use the name *pa* for this expression, through the assignment

pa := **agent** *c*. *p* (*t1*, *u1*)

where *pa* is of a *PROCEDURE* type (with generic parameters, to be introduced below), we can later on obtain the effect of an original call by executing

pa.*call* ([ ])

Here *call* is a feature of class *PROCEDURE*, which performs a call to the routine associated with

an agent. Feature *call* always takes a single argument, a tuple. (Tuples, discussed in the last column, represent sequences of values.) Here the argument is the empty tuple [ ]; this is because when defining the agent expression **agent** *c1*.*p* (*t1*, *u1*) we included a target *c1* and all the necessary arguments to *p* — *t1* and *u1* —, so at the time of the call to *call* we don't need any more arguments. We say that the agent expression is **closed** on all of its operands (target and arguments).

It is important to understand what *pa* represents: an object embodying all the properties of the procedure *p*, ready to be applied to the given target and arguments. C++ programmers might at first think of it as a "function pointer", but it is really much more: an object representing the procedure's properties. We can do many things with such an object: not just pass it around and call the associated routine (through procedure *call*), as we could do with a function pointer, but also obtain information about it, such as the class to which it belongs (here *C*), its possible redefinitions, its precondition, postcondition and so on. The last examples show how the agent mechanism opens up a whole set of **introspection** mechanisms.

If instead of starting from a procedure *p* you build an agent expression from a function *f* (*x*: *T* ; *y*: *U*): *V*, as in

fa := **agent** *c1*.*f* (*t1*, *u1*)

then in addition to *call* you can use on *fa* the feature *item*, such that *fa*.*item* gives the result returned by the last call to *fa*.*call* ([ ]); you can also combine *call* and *item* by using *fa*.*value* ([ ]), which returns the result of calling *f* on the given target and arguments.

All these operations can be applied at any time, by any routine that has access to the agent (*pa* or *fa* in these examples). By passing an agent expression to another unit, you enable it to call the corresponding routine (*p* or *f* here) whenever it pleases. So we may view an agent as a **delayed call**. Of course you can, as noted, do more with an agent than just call the associated routine.

Keeping arguments open

In the two examples so far, all operands (arguments and target) were closed — set at the time of the definition of the agent expression. In many cases you will want to leave some of the operands unspecified, or "open", in the agent, to be specified only at the time of the actual call as executed by feature *call*. The mechanism leaves you complete freedom as to which operands you choose to close and which you leave open.

To leave an argument open, simply replace it by a question mark. For example you can define the agents

pb := **agent** *c1*.*p* (**?**, *u1*)

pc := **agent** *c1*.*p* (*t1*, **?**)

pd := **agent** *c1*.*p* (**?**,**?**)

The first and second are open on one argument, the third is open on both arguments. With *t2* of type *T* and *u2* of type *U*, they can be used in calls such as

pb.*call* ([*t2*])

pc.*call* ([*u2*])

pd.*call* ([*t2, u2*])

Note how, in each case, the tuple must provide the arguments corresponding to the open positions in the defining agent expression. The first call will yield the same result as if, using the original procedure *p*, you had directly called *c1*. *p* (*t2, u1*); the second, *c1*.*p* (*t1, u2*); the third, *c1*. *p* (*t2, u2*). (Readers familiar with lambda calculus may think of "open" as free and "closed" as bound.)

As a syntactical facility, you may omit the argument list if all arguments are open. So the expression defining *pd* could also be written as just **agent** *c1* ~ *p*.

Along with operands, you may want to leave the **target** of a call open. In all previous examples this target, *c1*, was closed. To leave it open, the question mark notation would not work, because you need to state the type of the target. (This is not necessary for arguments, since once we know *c1* and its type we — and the compiler — know the types for the arguments of procedure *p* in the corresponding class *C*.) Instead of a question mark you will use the notation {*C*}, where *C* is the corresponding type, here class *C*. The following are agent expressions open on their targets:

pe := **agent** {*C*}. *p* (*t1*, *u1*) -- Open on its target only

pf := **agent** {*C*}. *p* (**?**, *u1*) -- Open on its target and first argument

pg := **agent** {*C*}. *p* (**?**,**?**) -- Open on all operands;

-- can also be written as just **agent** {*C*}. *p*

Corresponding routine calls will be of the following form (for *c2* of type *C*):

pe. *call* ([*c2*])

pf. *call* ([*c2*, *t2*])

pg. *call* ([*c2*, *t2*, *u2*])

Note how the arguments to *call* correspond to the open operands in the defining agent expression, without any distinction between target and argument. This property will be particularly important when we use agents to define iterators: we can use the same iterator mechanism to apply an operation repetitively to every element of a structure, whether the operation applies to its target or to its arguments.

For consistency and expressiveness, you can in fact use both of the open operand notations — question mark, and explicit type {*T*} in braces — for arguments as well as for the target.

The procedure *call* will always apply dynamic binding, that distinctive feature of object-oriented computation: the feature version to be applied is the one deduced, at run time, from the actual type of the target. This is particularly significant in the case of open targets (and is another difference with mere function pointers).

The types of agents

As noted in the introduction, the agent mechanism is type-safe. An agent is an instance of one of the new Kernel Library classes *PROCEDURE* and *FUNCTION*, both inheriting from *ROUTINE* (where routines such as *call* are introduced). These are generic classes; for example *FUNCTION* is declared as *FUNCTION* [*BASE*, *OPEN –> TUPLE*, *RES*]. The first generic parameter, *BASE*, represents the class to which the underlying routine belongs, *C* in our examples. *OPEN* represents the tuple of types of open operands; the Eiffel notation *–> TUPLE* indicates that *OPEN* is a "generically constrained" parameter, representing a type that must inherit from *TUPLE*, i.e. a tuple type. *RES* represents the type of the function result. Similarly, *ROUTINE* is declared as *ROUTINE* [*BASE*, *OPEN –> TUPLE*], with no result type; *PROCEDURE* is declared like *ROUTINE*.

Here are the types of the preceding example agents:

pa: *PROCEDURE* [*C*, *TUPLE*]
pb: *PROCEDURE* [*C*, *TUPLE* [*T*]]
pc: *PROCEDURE* [*C*, *TUPLE* [*U*]]
pd: *PROCEDURE* [*C*, *TUPLE* [*T*, *U*]]
pe: *PROCEDURE* [*C*, *TUPLE* [*C*]]
pf: *PROCEDURE* [*C*, *TUPLE* [*C*, *T*]]
pg: *PROCEDURE* [*C*, *TUPLE* [*C*, *T*, *U*]]

Note again how the *TUPLE* type used for the second parameter represents the open arguments, no difference being made between the target and the arguments. Correspondingly, procedure *call* is declared in class *ROUTINE* (and inherited by *PROCEDURE* and *FUNCTION*) with the signature

call (*arguments*: *OPEN*)

meaning that it takes a single argument, which must be a tuple of the type *OPEN* representing the open arguments.

Example uses

The above describes the essential properties of agents (the only significant missing part is the set of features in *ROUTINE* and related classes that will provide introspection facilities). Here are a few examples showing the versatility of the mechanism.

Assume you want to integrate a function *g* (*x: REAL*)*: REAL* over the interval [*0, 1*]. With *your_integrator* of a suitable type *INTEGRATOR* you will simply write the expression

your_integrator.*integral* (**agent** *g* (**?**), *0.0*, *1.0*)

A nice touch is that you can use exactly the same mechanism to integrate a function *h* of, say, three arguments, along its first argument: just use

your_integrator.integral (**agent** *h* (**?**, *u*, *v*), *0.0*, *1.0*)

Here is the way *integral* could look like (using an unsophisticated integration algorithm) in *INTEGRATOR*:

```
integral (f: FUNCTION [ANY, TUPLE [REAL], REAL];
          low, high: REAL): REAL is
          -- Integral of f over the interval [low, high]
     require
          meaningful_interval: low <= high
     local
```

```
        x: REAL
    do
        from x := low invariant
            x >= low ; x <= high + step
            -- Result approximates the integral over
            -- the interval [low, low.max (x – step)]
        until x > high loop

            Result := Result + step  * f.value ([x])
            x := x + step
        end
    end
```

Similar applications abound in numerical computation, and future editions of the EiffelMath library [1] may be able to benefit from the agent mechanism.

Consider now an integration example. Assume that in a class *C* we have a boolean-valued function *is_positive* that tells us whether a certain integer is positive:

```
is_positive (x: INTEGER) is do Result := (x > 0) end
```

and a list of integers *intlist*: *LINKED_LIST* [*INTEGER*]. With a suitable iterator function *for_all* in class *LIST* (inherited by *LINKED_LIST* and other variants) we can determine whether all elements in the list are positive:

```
all_positive := intlist.for_all (agent is_positive (?))
```

Remarkable here is the possibility to use the same iterator mechanism, *for_all*, to iterate a function that operates on its **target**, rather than on its argument as *is_positive* does. Consider a class *CUSTOMER* with a boolean-valued function *has_maintenance*; a typical call would be

this_customer.*has_maintenance*, operating on its target *this_customer* and returning a boolean value. We might have a list *customer_list*: *ARRAYED_LIST* [*CUSTOMER*]. To determine whether every element satisfies the property, we can just write:

```
all_on_maintenance := customer_list.for_all
                            (agent {CUSTOMER}.has_maintenance)
```

To switch from iterating an operation working on its argument to one working on its target, it suffices to make the target, rather than the argument, open. This is what the last expression does.

Here now is how the *for_all* iterator itself might look. It is just one of the iterators which will be added to the EiffelBase classes describing sequential structures; others are *do_all*, *do_while* and

so on. Recall that in these classes you can use *start* to move the cursor to the first element, *forth* to advance it by one position, *item* to access the element at cursor position, and *off* to find out if you have passed the last element:

```
do_all (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]) is
        -- Does every element of the structure satisfy test?
    require
        … Appropriate preconditions …
    do
        from
            Result := True ; start
        until off or not Result loop
            Result := test.value ([item])
            forth
        end
    end
```

(Sidebar) An exercise: once per object

To test your understanding of the agent mechanism (and other object-oriented techniques), here is a little exercise to which agents provide a neat solution. That solution will be published in the next installment of the Eiffel column.

Eiffel provides a powerful "once" mechanism [2]: if you declare a function (or other routine) using the **once** keyword rather than the more common **do**, the body will be executed only the first time around. So by writing a class *C* that includes a function

```
error_window: WINDOW is
        -- Window where error messages will be displayed
    once
        create Result.make (x, y, height, width)
    ensure
        exists: Result /= Void
    end
```

you ensure that the body (the **create** instruction) will be executed only once. Subsequent calls will return immediately; for a function, as here, the result will always be the one computed by the first call (the window that this first call will have created). Using **do** rather than **once** would not produce the desired result, since every call would produce a new window. Here we want *error_window* to denote the same window in every instance

of *C*.

This is a "once per class" mechanism. In some cases you might want to achieve the effect of "**once per objec**t": a routine that is executed only the first time it is called on any particular object. As an example, consider a class *TRADED_COMPANY*, representing companies traded on, say, Nasdaq. One of the features of the class is *history*, of type *COMPANY_HISTORY*, representing historical data on the company. The company history is stored in a database, however, so evaluating *company* requires loading large amounts of data into memory. We don't want to do that whenever an instance of *TRADED_COMPANY* is created, because this would require loading huge amounts of data — potentially, in fact, the whole database of histories of all companies! Instead, we want to create a *COMPANY_HISTORY* object when and only when feature *company* is evaluated on a particular object. This is a typical "once per object" (not per class) situation.

The exercise is: devise a "once per object" mechanism that will be easy to use in any such situation. You may restrict yourself to functions (such as *company*), but otherwise the solution must be completely general. *A hint*: since the mechanism must cover functions returning results of an arbitrary type, you can use Eiffel's genericity to provide the required flexibility.

Status and current applications

Together with the extensions discussed in the previous column, the agent mechanism (introduced in 4.3) is available in the current released version of ISE Eiffel, 4.4. It is submitted for standardization. A complete definition of the mechanism may be found at http://www.eiffel.com (follow "papers").

All the facilities described in this paper are supported. (We have alluded to the future introspection facilities of class *ROUTINE*; this is the only part not available at the time of writing.) The mechanism has already been applied widely by users, and the response has been enthusiastic. Within ISE it is also used in critical applications; in particular the command mechanism, so important for interactive applications [3], makes extensive use of agents in the newest version of the EiffelVision portable graphical library. The introduction of agents in no way invalidates command classes and the object-oriented principles that stand behind them; in fact they reinforce these techniques, while simplifying their application and reducing the number of necessary command classes.

In a similar fashion, AxaRosenberg, in its investment software, used to have hundreds of separate classes to represent the so-called *fundamental variables* of every company being tracked by the system. Thanks to agents, these classes have been reduced to just a few, with agent expressions taking care of the specifics of each company.

Although not yet part of official Eiffel, agents nicely complement the existing mechanisms, and all indications are that they are here to stay.

References

[1] *Object Technology for Scientific Computing —- Object-Oriented Numerical Software in Eiffel and C*, Paul Dubois, Prentice Hall, 1996.

[2] *Eiffel: The Language*, Bertrand Meyer, Prentice Hall, 1991 (second printing).

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

Paul Dubois is a Team Leader at Lawrence Livermore National Laboratory and Scientific Programming Editor, IEEE *Computers in Science and Engineering*. Mark Howard is Associate Director of Barr Rosenberg Research Center LLC, an investment research firm in Orinda, California. Bertrand Meyer is president of Interactive Software Engineering, and an adjunct professor at Monash University; he is the editor of the Eiffel column and you can send comments to him at Bertrand.Meyer@eiffel.com. Michael Schweitzer and Emmanuel Stapf are members of the Eiffel compiler team at Interactive Software Engineering.