CHAPTER THIRTEEN

Software Architecture: Object-Oriented Versus Functional

Bertrand Meyer

One of the arguments for functional programming is better modular design. By analyzing publications advocating this approach, in particular through the example of a framework for financial contracts, we access its strengths and weaknesses, and compare it with object-oriented design. The overall conclusion is that object-oriented design, especially in a modern form supporting high-level routine objects or "agents," *subsumes* the functional approach, retaining its benefits while providing higher-level abstractions more supportive of extension and reuse.

Overview

"Beauty," as a slogan for a software architecture, is not strictly for the beholder to judge. Clear objective criteria exist (Meyer 1997):

Reliability
: Does the architecture help establish the correctness and robustness of the software?

Extendibility
: How easy is it to accommodate changes?

Reusability

Is the solution general, or better yet, can we turn it into a *component* to be plugged in directly, off-the-shelf, into a new application?

The success of object technology has largely followed from the marked improvements it brings—if applied properly as a method, not just through the use of an object-oriented programming language—to the reliability, extendibility, and reusability of the resulting programs.

The *functional programming* approach predates object-oriented thinking, going back to the Lisp language available for almost 50 years. To those fortunate enough to have learned it early, functional programming will always remain like the memory of a first kiss: sweet, and the foretaste of even better experiences. Functional programming has made a comeback in recent years, with the introduction of new languages such as Scheme, Haskell, OCaml and F#, sophisticated type systems, and advanced language mechanisms such as monads. Functional programming even seems at times to be presented as an improvement over object-oriented techniques. The present discussion compares the two approaches, using the cited software architecture criteria. It finds that the relationship is the other way around: object-oriented architecture, particularly if enriched with recent developments such as *agents* in Eiffel terminology ("closures" or "delegates" in other languages), subsumes functional programming, retaining its architectural advantages while correcting its limitations.

To qualify this finding, it is important to note both the study's limitations and arguments to mitigate some of them. The limitations include:

Few data points

The analysis is primarily based on two examples of functional design. This could cast doubts on the generality of the lessons drawn.

Lack of detail

The source of the examples consists of an article (Peyton Jones et al. 2000) and a PowerPoint presentation (Eber et al. 2001)—referred to from now on as "the article" and "the presentation"—complemented in the section "Assessing the Modularity of Functional Solutions," later in this chapter, by ideas from a classic functional programming paper (Hughes 1989). Uses of the presentation may miss some details and nuances that would be present in a more discursive document.

Specific focus

We only consider the issue of modularity. The case for functional programming also relies on other criteria, such as the elegance of a declarative approach.

Experimenter bias

The author of the present chapter is a long-time contributor to and exponent of object technology.

The following observations counterbalance some of this possible criticism:

- The functional examples come from industrial practice; specifically, a company whose business appears to rest on the application of functional programming techniques. The principal example—specifying sophisticated financial instruments—addresses complex problems faced by the financial industry, which current tools do not address well according to the presentation's author, an expert in that industry. This suggests that it is representative of the state of the art. (The first example—specifying puddings—is academic, intended only as a pedagogical stepping stone.)
- One of the authors of the article (S. Peyton Jones), also acknowledged in the presentation as coauthor of the underlying theoretical work, is the lead designer of the Haskell language and one of the most notable figures in functional programming, bringing considerable credibility. The paper used as a subsidiary example in the later section "Assessing the Modularity of Functional Solutions" has been extremely influential and was written by another leading member of the functional programming community (J. Hughes).
- In spite of the reservations expressed below, the solutions described in these documents are elegant and clearly the result of considerable reflection.
- The examples do not exercise the notion of changeable *state*, which would favor an imperative object-oriented programming style.

We must also note that mechanisms such as agents, which provide essential ingredients of the full object-oriented solution, were openly inspired by functional programming ideas. So the conclusion will not be a dismissal of the functional school's contribution, simply the observation that the object-oriented (OO) style is more suited for defining the overall architecture of reliable, extendible, and reusable software, while the building blocks may involve a combination of OO and functional techniques.

Further observations about the following discussion:

- Object technology as used here takes the form of Eiffel. We have not attempted to analyze what remains if one *removes* mechanisms such as multiple inheritance (absent in Java and C#), genericity (absent in earlier versions of these languages), contracts (absent outside of Eiffel except in JML and Spec#), or agent-style facilities (absent in Java), or if one *adds* mechanisms such as overloading and static functions, which threaten the solidity of the OO edifice.
- The discussion is about architecture and design. In spite of its name, functional programming is (like object technology) relevant to these tasks and not just to "programming" in the restricted sense of implementation. The Eiffel approach explicitly introduces a continuum from specification to design and implementation through the concept of seamless development. Implementation-oriented properties of either approach, while important in practice, will not be considered in any detail.

- Also relevant in practice are issues of expressiveness and notation. They are taken into account to the extent that they affect the key criteria of architecture and design. For the most part, however, the discussion considers semantics rather than syntax.

Two more preliminary notes. First, terminology: by default, the term "contract" refers to financial contracts, relevant to the application domain of the article and presentation, and not to be confused with the software notion of Design by Contract* (the idea [Meyer 1997] of including elements of specification such as preconditions, postconditions, or invariants). In case of possible ambiguity, the terms used here will be *financial contracts* and *software contracts*.

Second, a semi-apology: when the discussion moves to OO territory in its second half, it includes more references to and repetitions from the author's previous publications than discretion would command. The reason is that the wide spread of object technology has been accompanied by the loss of some of its more subtle but (in our opinion) critical principles, such as command-query separation (see "State Intervention" later in this chapter); this makes some brief reminders necessary. For the full rationale behind these ideas, see the cited references.

The Functional Examples

The overall goal of the article and presentation is to propose a convenient mechanism for describing and handling financial contracts, especially modern financial instruments that can be very complicated, as in this example from the presentation (in whose numerical values one can hear the nostalgic echo of a time when major currencies enjoyed a different relationship):

> "Against the promise to pay USD 2.00 on December 27 (the price of the option), the holder has the right, on December 4, to choose between:
>
> - Receiving USD 1.95 on December 29, or
> - Having the right, on December 11, to choose between:
>   - Receiving EUR 2.20 on December 28, or
>   - Having the right, on December 18, to choose between:
>   - Receiving GBP 1.20 on December 30, or
>   - Paying immediately one more EUR and receiving EUR 3.20 on December 29"

(Throughout this section, extracts in quotes are direct citations from the presentation or the article. Elements not in quotes are our interpretations and comments.)

As a pedagogical device to illustrate the issues, the presentation starts with a toy example: *puddings* rather than contracts. From the precise description of a pudding, it should be possible

* Design by Contract is a trademark of Eiffel Software.

to “compute the sugar content,” “estimate the time to make” the pudding, and obtain “instructions to make it.” A “bad approach” would be to:

- “List all puddings (Trifle, lemon upside-down pudding, Dutch apple cake, Christmas pudding)
- For each pudding, write down sugar content, time to make, instructions, etc.”

Although the presentation does not state why the approach is bad, we can easily surmise the reasons: as a collection of ad hoc descriptions, it has no reusability, since it does not take advantage of the property that different kinds of pudding may share the same basic parts; it has no extendibility, since any modification of a pudding part will require reworking all the puddings that rely on that part.

The pudding is a metaphor for the examples of real interest, contracts, but since it is easily understandable without a specialized knowledge domain, we continue with it. A “good approach” is to:

- “Define a small set of ‘pudding combinators.’
- Define all puddings in terms of these combinators.
- Calculate sugar content from these combinators too.”

A combinator is an operator that produces a composite object from similar objects. The tree shown in Figure 13-1, from the presentation, illustrates what the combinators may be in this example.

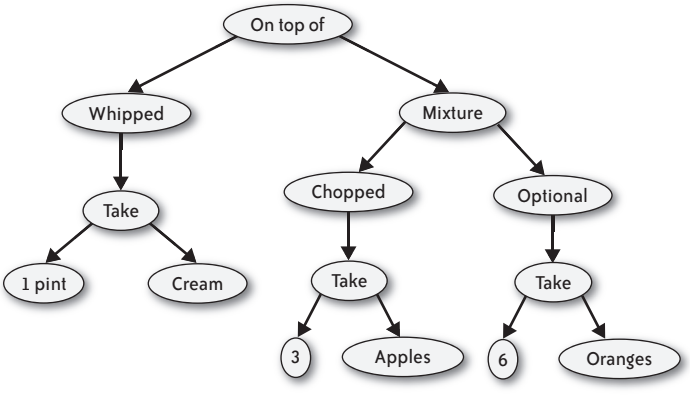

FIGURE 13-1. Ingredients and combinators describing a pudding recipe

> **NOTE**
>
> **We share the reader's alarm at the unappetizing nature of the example, especially coming from a Paris-based author. The sympathetic explanation is that the presentation was directed to a foreign audience of which it assumed, along with unfamiliarity with the metric system, barbaric culinary habits. The present discussion relies on the assumption that bad taste in desserts is not a sufficient predictor of bad taste in language and architecture paradigms.**

The nonleaf nodes of the tree represent combinators, applied to the subtrees. For example, "Take" is a combinator that assumes two arguments, a pudding part ("Cream" on the left, "Oranges" on the right) and a quantity ("1 pint" and "6"); the result of the application, represented by the tree node, is a pudding part or a pudding made of the given quantity of the given part.

It is also possible to write out such a structure textually, using a mini-"domain-specific language" (DSL) "for describing puddings" (boldface is added for operators):

```
"salad      = on_top_of topping main_part"      -- Changed from
        "OnTopOf" for consistency
"topping    = whipped (take pint cream)
main_part   = mixture apple_part orange_part
apple_part  = chopped (take 3 apples)
orange_part = optional (take 6 oranges)"
```

This uses an anonymous but typical—the proper term might be "vanilla"—variant of functional programming notation, where function application is simply written as function args (for example, `plus a b` for the application of `plus` to `a` and `b`) and parentheses serve only for grouping.

With this basis, it becomes a piece of cake to define an operation such as sugar content (`S`) by case analysis on the combinators (similar to defining a mathematical function on recursively defined objects by using a definition that follows the same recursive structure):

```
"S  (on_top_of p1 p2)   = S (p1) + S (p2)
S   (whipped p)         = S (p)
S   (take q i)          = q * S(i)
etc."
```

Not clear (to us) from the "etc." is how operators such as `S` deal with the `optional` combinator; there has to be some way of specifying whether a particular concoction has the optional part or not. This issue aside, the approach brings the benefits that the presentation claims for it:

- "When we define a new recipe, we can calculate its sugar content with no further work.
- Only if we add new combinators or new ingredients would we need to enhance `S`."

The real goal, of course, is not pudding but contracts. Here the presentation contains a sketch of the approach but the article is more detailed. It relies on the same ideas, applied to a more interesting set of elements, combinators, and operations.

The *elements* are financial contracts, dates, and observables (such as a certain exchange rate on a certain date). Examples of basic contracts include `zero` (can be acquired at any time, no rights, no obligations) and `one (c)` for a currency `c` (immediately pays the holder one unit of `c`).

Examples of *combinators* on contracts include: `or`, such that acquiring the contract (`or c1 c2`) means acquiring either of `c1` and `c2`, and expiring when both have expired; `anytime`, such that (`anytime c`) can be acquired at any time before the expiration of `c`, and expiring whenever `c` expires; `truncate`, such that (`truncate t c`) is like `c` except that it expires at the earlier of `t` and the expiry of `t`; and `get`, so that acquiring (`get c`) means acquiring `c` at its expiry date. The paper lists about a dozen such basic combinators on contracts, and others on observables and dates. They make it possible to define advanced financial instruments such as a "European option" in a simple way:

```
european t u   = get (truncate t (or u zero))
```

Operations include the expiry date of a contract and—the most important practical benefit expected from all this modeling effort—its value process, a time-indexed sequence of expected values. As with the sugar content of a pudding, the functions are defined by case analysis on the basic constructors. Here are the cases involving the preceding basic elements and combinators for the operation `H`, which denotes the expiry date or "horizon":

```
H (zero)         =  ∞   -- Where ∞ is a special value with the
       expected properties
H (or c1 c2)     =  max (H (c1), H (c2))
H (anytime c)    =  H (c)
H (truncate t c) =  min (t, H (c))
H (get c)        =  H (c)
```

The rules yielding value processes follow a similar structure, although the righthand sides are more sophisticated, involving financial and numerical computations. For more examples of applying combinators and functional programming ideas to financial applications, see Frankau (2008).

Assessing the Modularity of Functional Solutions

The preceding presentation, while leaving aside many contributions of the presentation and especially the article, suffices as a basis for discussing architectural features of the functional approach and comparing them with the OO view. We will freely alternate between the pudding example (which makes the ideas immediately understandable) and financial contracts (representative of real applications).

Extendibility Criteria

As pointed out by the presentation, the immediate architectural benefit is that it is easy to add a new combinator: "When we define a new recipe, we can calculate its sugar content with no further work." This property, however, is hardly a consequence of using a functional programming approach. The insight was to introduce the notion of a combinator, which creates pudding and pudding parts—or contracts—from components that can either be atomic or themselves result from applying combinators to more elementary components.

The article and presentation suggest that this is a new idea for financial contracts. If so, the insights should be beneficial to financial software. But as a general software design idea, they are not new. Transposed to the area of GUI design, the "bad approach" rejected at the beginning of the presentation (list all pudding types, for each of them compute sugar content, etc.) would mean devising every screen of an interactive application in its own specific way and writing the corresponding operations—display, move, resize, hide—separately in each case. No one ever does this. Any GUI design environment provides atomic elements, such as buttons and menu entries, and operations to combine them recursively into windows, menus, and other containers to make up a complete interface. Just as the pudding combinators define the sugar content and calorie count of a pudding from those of its ingredients, and contract combinators define the horizon and value sequence of a complex contract from those of its constituents, the display, move, resize, and hide operations on a composite figure apply these operations recursively on the components. The EiffelVision library (see the EiffelVision documentation at *http://eiffel.com*) is an example application of this compositional method, systematic but hardly unique. The article's contribution here is to apply the approach to a new application area, financial contracts. The approach itself, however, does not assume functional programming; any framework with a routine mechanism and recursion will do.

Interesting modularity issues arise not when existing combinators are applied to components of existing types, but when the combinators and component types change. The presentation indeed states: "Only if we add new combinators or new ingredients would we need to enhance `S`" (the sugar combinator). The interesting question is how disruptive such changes will be to the architecture.

The set of relevant changes is actually larger than suggested:

- Along with *atomic types* and *combinators*, we should consider changes in *operations*: adding a calorie count function for puddings, a delay operation for contracts, and a rotate operation for graphical objects.
- Besides such additions, we should include *changes* and *removal*, although for simplicity this discussion will continue to consider additions only.

Assessing the Functional Approach

The structure of the programs as given is simple—a set of definitions of the form:

```
0 (a)             = b_a,0                    [1]
0 (c (x, y, ...)  = f_c,0 (x, y, ...)        [2]
```

for every operation `0`, atomic type `a`, and basic combinator `c`. The righthand sides involve appropriate constants `b` and functions `f`. Again for simplicity, we may view the atomic types such as `a` as 0-ary combinators, so that we only need to consider form [2]. With `t` basic combinators (`on_top_of`, `hipped`…) and `f` operations (sugar content, calories), we need `t` × `f` definitions.

Regardless of the approach, these `t` × `f` elements will have to be accommodated. The architectural problem is how we group them into modules to facilitate extension and reuse. This issue is not discussed in the article and presentation. Of course, the matter is not critical for small `t` and `f`; then all the definitions can be packed into a single module. This takes care of extendibility in a simple way:

- To add a basic combinator `c`, add `f` definitions of the above form, one for each existing operation.
- To add an operation `O`, add `t` definitions, one for each existing combinator.

This approach does not scale well; for larger developments, it will be necessary to divide the system into modules; the extendibility problem then becomes how to make sure that such modifications affect as few modules as possible.

Even with fairly small `t` and `f`, the one-module solution does not support reusability: if another program only needs a subset of the operations and combinators, it would suffer the usual dilemma of primitive modularization techniques:

Charybdis
: Copy-paste the relevant parts, but then risk forgetting to update the derived modules when something changes in the original (possibly for such a prosaic reason as a bug fix).

Scylla
: Use a module inclusion facility, as provided by many languages, to make the contents of an existing module available to a new one; but you end up loaded with a bigger baggage than necessary, which complicates updates and may cause conflicts (assuming the derived module defines a new combinator or function and a later version of the original module introduces a clashing definition).

These observations remind us in passing that reusability is closely connected to extendibility. An online critique of the OCaml functional language (Steingold 2007) takes a concrete example:†

> You cannot easily modify the behavior of a module outside of it. Suppose you use a Time module defining `Time.date_of_string`, which parses ISO8601 basic format ("YYYYMMDD"), but want to recognize ISO8601 extended format ("YYYY-MM-DD"). Tough luck: you have to get the module maintainer to edit the original function—you cannot redefine the function yourself in your module.

As software grows and changes, another aspect of reuse becomes critical: reuse of common properties. Along with European options, the article introduces "American options." Described as combinators, they have different signatures (`Date → Contract → Contract` and `(Date, Date) → Contract → Contract`). One suspects, however, that the two kinds of option have a number of

† This citation is slightly abridged. Inclusion of the citation does not imply endorsement of other criticism on that page.

properties and operations in common, in the same way that puddings can be grouped into categories. Such groupings would help model and modularize the software, with the added benefit—if enough commonalities emerge—of reducing the number of required definitions. This requires, however, taking a new look at the problem domain: we must discover, beyond functions, the essential *types*.

Such a view will be at a higher level of abstraction. One can argue in particular with the fixation on functions and their signatures. According to the article (italics retained), "An *American option* offers more flexibility than a European option. Typically, an American option confers the right to acquire an underlying contract *at any time between two dates*, or not to do so at all." This suggests a definition by variation: either American options are a special case of European option, or they are both variants of a more general notion of option. Defining them as combinators immediately sets them apart from each other because of the extra Date in the signature. This is akin to defining a concept by its implementation—a mathematical rather than computer implementation, but still implying loss of abstraction and generality. Using types as the basic modularization mechanism, as in object-oriented design, will elevate the level of abstraction.

Levels of Modularity

Assessing functional programming against criteria of modularity is legitimate since better modularization is one of the main arguments for the approach. We have seen the presentation's comments on this issue, but here is a more general statement from one of the foundational papers of functional programming, by Hughes (1989), stating that with this approach:

> [Programs] can be modularized in new ways, and thereby greatly simplified. This is the key to functional programming's power—it allows greatly improved modularization. It is also the goal for which functional programmers must strive—smaller and simpler and more general modules, glued together with the new glues we shall describe.

The "new glues" described in Hughes's paper are the ones we have seen at work for the two examples covered—systematic use of stateless functions, including high-level functions (combinators) that act on other functions—plus the extensive use of lists and other recursively defined types, and the concept of lazy evaluation.

These are attractive techniques, but they address fine-grain modularization. Hughes develops a functional version of the Newton-Raphson computation of the square root of a number `N` with tolerance `eps` and initial approximation `a0`:

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

with appropriate combinators `within`, `repeat`, and `next`, and compares this version with a FORTRAN program involving `goto` instructions. Even ignoring the cheap shot (at the time of the paper's original publication, FORTRAN was already old hat and `goto`s despised), it is understandable why some people prefer such a solution, based on small functions glued

through combinators, to the loop version. Then again, others prefer loops, and because we are talking about the fine-grain structure of programs rather than large-scale modularization, the issue hardly matters for software engineering; it is a question of style and taste. The more fundamental question of demonstrating correctness has essentially the same difficulty in both approaches; note, for example, that the definition of `within` in Hughes's paper, yielding the first element of a sequence that differs from the previous one by less than `eps`:

```
within eps ([a:b:rest])  =  if abs (a - b) <= eps then b
else within eps [b:rest]
```

seems to assume that the distances between adjacent elements are decreasing, and definitely assumes that one of these differences is no greater than `eps`.‡ Stating this property would imply some Design by Contract-like mechanism to associate preconditions with functions (there is no such mechanism in common functional approaches); the proof that it guarantees termination of `eps` would be essentially the same as a proof of termination for the corresponding loop in the imperative style.

There seems to be no contribution to large-grain modularity or software architecture in this and earlier examples. In particular, the stateless nature of functional programming does not seem (positively or negatively) to affect the issue.

The Functional Advantage

There remains four significant advantages for the functional approach as illustrated in examples so far.

The first is notational. No doubt some of the attraction of functional programming languages comes from the terseness of definitions such as the above. This needs less syntactical baggage than routine declarations in common imperative languages. Several qualifications limit this advantage:

- In considering design issues, as in the present discussion, the notational issue is less critical. One could, for example, use a functional approach for design and then target an imperative language.
- Many modern functional languages such as Haskell and OCaml are strongly typed, implying the notation will be a little more verbose; for example, unless the designer wants to rely on type inference (not a good idea at the design stage), `within` needs the type declaration `Double → [Double] → Double`.
- Not everyone may be comfortable with the common practice of replacing multiargument functions by functions returning functions (known in the medical literature as RCS, for "Rabid Currying Syndrome," and illustrated by such signatures as `(a → b → c) → Obs a → Obs b → Obs c` in the financial article). This is a matter of style rather than a fundamental

‡ In citing examples from Hughes's paper we have, with his agreement, used modern (Haskell) notation for lists, as in `[a:b:rest]`, more readable than the original's `cons` notation, as in `cons a (cons b rest)`.

property of the approach, which does not require it, but it is pervasive in these and many other publications.

Still, notation conciseness is a virtue even at the design and architecture level, and functional programming languages may have some lessons here for other design notations.

The second advantage (emphasized by Simon Peyton Jones and Diomidis Spinellis in comments on an earlier version of this chapter), also involving notation, is the elegance of combinator expressions for defining objects. In an imperative object-oriented language, the equivalent of a combinator expression, such as:

```
on_top_of topping main_part
```

would be a creation instruction:

```
create pudding .make_top (topping, main_part)
```

with a creation procedure (constructor) `make_top` that initializes attributes `base` and `top` from the given arguments. The combinator form is descriptive rather than imperative. In practice, however, it is easy and indeed common to use a variant of the combinator form in object-oriented programming, using "factory methods" rather than explicit creation instructions.

The other two advantages are of a more fundamental nature. One is the ability to manipulate operations as "first-order citizens"—the conventional phrase, although we can simply say "as objects of the program" or just "as data." Lisp first showed that this could be done effectively; a number of mainstream languages offered a way to pass routines as arguments to other routines, but this was not considered a fundamental design technique, and was in fact sometimes viewed with suspicion as reminiscent of self-modifying code with all the associated uncertainties. Modern functional languages showed the benefit of accepting higher-order functionals as regular program objects, and developed the associated type systems. This is the part of functional programming that has had the most direct effect on the development of mainstream approaches to programming; as will be seen below, the notion of `agent`, directly derived from these functional programming concepts, is a welcome addition to the original object-oriented framework.

The fourth significant attraction of functional programming is lazy evaluation: the ability, in some functional languages such as Haskell, to describe a computation that is potentially infinite, with the understanding that any concrete execution of that computation will be finite. The earlier definition of `within` assumes laziness; this is even more clear in the definition of `repeat`:

```
repeat f a  = [a : repeat f (f a)]
```

which produces (in ordinary function application notation) the infinite sequence `a`, `f (a)`, `f (f (a))`.... With `next N x` defined as `(x + N / x) / 2`, the definition of `within` as used by `sqrt` will stop evaluating that sequence after a finite number of elements.

This is an elegant idea. Its general application in software design calls for two observations.

First, there is the issue of correctness. The ease of writing potentially infinite programs may mask the difficulty of ensuring that they will always terminate. We have seen that `within` assumes a precondition, not stated in its presentation; this precondition, requiring that elements decrease to below `eps`, cannot be finitely evaluated on an infinite sequence (it is semi-decidable). These are tricky techniques for designers to use, as illustrated by the problem of how many lazy functional programmers it takes to change a light bulb. (It is hard to know in advance. If there are any lazy functional programmers left, ask one to change the bulb. If she fails, try the others.)

Second and last, lazy manipulation of infinite structures is possible in a nonfunctional design environment, without any special language support. The abstract data type approach (also known as object-oriented design) provides the appropriate solution. Finite sequences and lists in Eiffel libraries are available through an API relying on a notion of "cursor" (see Figure 13-2).

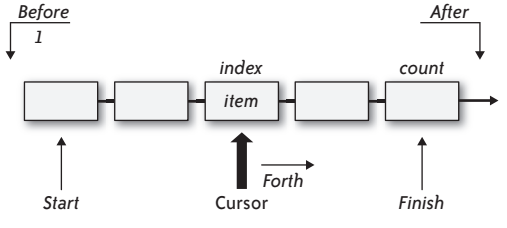

FIGURE 13-2. Cursors in Eiffel lists

Commands to move the cursor are `start` (go to first item), `forth` (move to next item), and `finish`. Boolean queries `before` and `after` tell if the cursor is before the first element or after the last. If neither holds, `item` returns the element at cursor position, and `index` its index.

It is easy to adapt this specification to cover infinite sequences: just remove `finish` and `after` (as well as `count`, the number of items). This is the specification of the deferred (abstract) class `COUNTABLE` in the Eiffel library. Some of its descendants include `PRIMES`, `FIBONACCI`, and `RANDOM`; each provides its implementations of `start`, `forth`, and `item` (plus, in the last case, a way to set the seed for the pseudo-random number generator). To obtain successive elements of one of these infinite sequences, it suffices to apply `start` and then query for `item` after any finite number of applications of `forth`.

Any infinite sequential structure requiring finite evaluation can be modeled in this style. Although this does not cover all applications of lazy evaluation, the advantage is to make the infinite structure explicit, so that it is easier to establish the correctness of a lazy computation.

State Intervention

The functional approach seeks to rely directly on the properties of mathematical functions by rejecting the assumption, present but implicit in imperative approaches, that computing

operations can, in addition to delivering a result (as a mathematical function does), modify the *state* of the computation: either a global state or, in a more modular approach, some part of that state (for example, the contents of a specific object).

Although prominent in all presentations of functional programming, this property is not visible in the examples discussed here, perhaps because they follow from an initial problem analysis that already whisked the state away in favor of functional constructs. It is possible, for example, that a nonfunctional model of the notion of valuation of a financial contract would have used, instead of a function that yields a sequence (the value process), an operation that transforms the state to update the value.

It is nevertheless possible to make general comments on this fundamental decision of functional approaches. The notion of state is hard to avoid in any model of a system, computerized or not. One might even argue that it is the central notion of computation. (It has been argued [Peyton Jones 2007] that stateless programming helps address issues of concurrent programming, but there is not enough evidence yet to draw a general conclusion.) The world does not clone itself as a result of each significant event. Neither does the memory of our computers: it just overwrites its cells. It is always possible to model such state changes by positing a sequence of values instead, but this can be rather artificial (as suggested by the alternative answer to the earlier riddle: functional programmers never change a bulb, they buy a new lamp with a new cable, a new socket, and a new bulb).

Recognizing the impossibility of ignoring the state for such operations as input and output, and the clumsiness of earlier attempts (Peyton Jones and Wadler 1993), modern functional languages, in particular Haskell, have introduced the notion of monad (Wadler 1995). Monads embed the original functions in higher-order functions with more complex signatures; the added signature components can serve to record state information, as well as any extra elements such as an error status (to model exception handling) or input-output results.

Using monads to integrate the state proceeds from the same general idea—used in the reverse direction—as the technique described in the last section for obtaining lazy behavior by modeling infinite sequences as an abstract data type: to emulate in a framework A a technique T that is *implicit* in a framework B, program in A an *explicit* version of T or of the key mechanism making T possible. T is infinite lists in the first case (the "key mechanism" is infinite lists evaluated finitely), and the state in the second case.

The concept of monad is elegant and obviously useful for semantic descriptions of programming languages (especially for the denotational semantics style). One may wonder, however, whether it is the appropriate solution as a mechanism to be used directly by programmers. Here we must be careful to consider the right arguments. The obvious objection to monads—that they are difficult to teach to ordinary programmers—is irrelevant; innovative ideas considered hard at the time of their introduction can fuse into the mainstream as educators develop ways to explain them. (Both recursion and object-oriented programming were once considered beyond the reach of "Joe the Programmer.") The important question is

whether this is worth the trouble. Making the state available to functional programmers through monads is akin to telling your followers, after you convinced them to embrace chastity, that having children is actually good, if with you.

Is it really necessary to exclude the state in the first place? Two observations are enough to raise doubts:

- *Elementary* state-changing operations, such as assignment of simple values, have a clear mathematical model (Hoare rules, based on substitution). This diminishes the main benefit expected of stateless programming: to facilitate mathematical reasoning about programs.
- For the more *difficult* aspects of establishing the correctness of a design or implementation, the advantage of the functional approach is not so clear. For example, proving that a recursive definition has specific properties and terminates requires the equivalent of a loop invariant and variant. It is also unlikely that efficient functional programs can afford to renounce programmer-visible linked data structures, with all the resulting problems such as aliasing, which are challenging regardless of the underlying programming model.

If functional programming fails to bring a significant simplification to the task of establishing correctness, there remains a major practical argument: referential transparency. This is the notion of substitutivity of equals for equals: in mathematics, `f (a)` always means the same thing for given values of `f` and `a`. This is also true in a pure functional approach. In a programming language where functions can have side effects, `f (a)` can return different results in successive invocations. Renouncing such possibilities makes it much easier to understand program texts by retaining the usual modes of reasoning from mathematics; for example, we are all used to accepting that `g + g` and `2 × g` have the same meaning, but this ceases to be guaranteed if `g` is a side effect-producing function. The difficulty here is not so much for automatic verification tools (which can detect that a function produces side effects) as for human readers.

Maintaining referential transparency in expressions is a highly desirable goal. It does not, however, justify removing the notion of state from the computational model. It is important to recall here the rule defined in the Eiffel method: *command-query separation principle* (Meyer 1997). In this approach the features (operations) of a class are clearly divided into two groups: commands, which can change the target objects and hence the state; and queries, which provide information about an object. Commands do not return a result; queries may not change the state—in other words, they satisfy referential transparency. In the above list example, commands are `start`, `forth`, and (in the finite case) `finish`; queries are `item`, `index`, `count`, `before`, and (finite case) `after`. This rule excludes the all-too-common scheme of calling a function to obtain a result *and* modify the state, which we guess is the real source of dissatisfaction with imperative programming, far more disturbing than the case of explicitly requesting a change through a command and then requesting information through a (side effect-free) query. The principle can also be stated as, "*Asking a question should not change the answer.*" It implies, for example, that a typical input operation will read:

```
io.read_character
Result:= io.last_character
```

Here `read_character` is a command, consuming a character from the input; `last_character` is a query, returning the last character read (both features are from the basic I/O library). A contiguous sequence of calls to `last_character` would be guaranteed to return the same result repeatedly. For both theoretical and practical reasons detailed elsewhere (Meyer 1997), the command-query separation principle is a methodological rule, not a language feature, but all serious software developed in Eiffel observes it scrupulously, to the benefit of referential transparency. Although other schools of object-oriented programming do not apply it (continuing instead the C style of calling functions rather than procedures to achieve changes), it is in our view a key element of the object-oriented approach. It seems like a viable way to obtain the referential transparency goal of functional programming—since expressions, which only involve queries, will not change the state, and hence can be understood as in traditional mathematics or a functional language—while acknowledging, through the notion of command, the fundamental role of the concept of state in modeling systems and computations.

An Object-Oriented View

We now consider how to devise an object-oriented architecture for the designs discussed in the presentation and article.

Combinators Are Good, Types Are Better

So far we have dealt with operations and combinators. Operations will remain; the key step is to discard combinators and replace them with types (or classes—the distinction only arises with genericity as discussed below). This brings a considerable elevation of the level of abstraction:

- A combinator describes a specific way of building a new mechanism from existing ones. The combination is defined in a rigid way: a `take` combination (as in `take` 3 apples) associates one quantity element and one food element. As noted earlier, this is the mathematical equivalent of defining a structure by its implementation.
- A class defines a type of objects by listing the applicable features (operations). It provides abstraction in the sense of abstract data types: the rest of the world knows the corresponding objects solely through the applicable operations, not from how they were constructed. We may capture these principles of data abstraction and object-oriented design by noting that the approach means knowing objects not from what they *are* but through what they *have* (their public features and the associated contracts). This also opens the way to taxonomies of types, or *inheritance*, to keep the complexity of the model under control and take advantage of commonalities.

By moving from the first approach to the second one, we do not lose anything, since classes trivially include combinators as a special case. It suffices to provide features giving the constituents, and an associated creation procedure (constructor) to build the corresponding objects. In the `take` example:

```
class REPETITION create
      make
feature
      base: FOOD
      quantity: REAL
      make (b: FOOD; q: REAL)
        -- Produce this food element from quantity units of base.
           ensure
                base = b
                quantity = q
           end
  ... Other features ...
end
```

This makes it possible to obtain an object of this type through `create apple_salad.make (6.0, apple)`, equivalent to an expression using the combinator. It is possible, as mentioned, to bring the notation closer to combinators by using factory methods.

Using Software Contracts and Genericity

Since we are concentrating on design, the effect of `make` has been expressed in the form of a postcondition, but it really would not be a problem to include the implementation clause (`do base := b; quantity := q`). It is one of the consequences of well-understood OO design to abate the distance between implementation and design (and specification). In all this we are freely using state-changing assignment instructions and still have (we thank the reader for inquiring) most of our teeth and hair.

Unlike the combinator, however, the class is not limited to these features. For example, it may have other creation procedures. One can usually mix two repetitions of the same thing:

```
make (r1, r2: REPETITION)
      -- Produce this food element by combining r1 and r2.
     require
          r1.base = r2.base
     ensure
          base = r1.base
          quantity = q
```

The precondition expresses that the quantities being mixed are from the same basic food types. This requirement can also be made static through the type system; genericity (also available in typed functional languages, under the curious if impressive-sounding name of "parametric polymorphism") leads to defining the class as:

```
class REPETITION  [FOOD] create
       ... As before ...
feature
      make (r1, r2: REPETITION [FOOD])
        ... No precondition necessary here ...
        ... The rest as before ...
end
```

Not only can classes have different creation procedures, they will generally have many more features. Specifically, the *operations* of our previous versions become features of the appropriate classes. (The reader may now have guessed that the variable name `t` stood for `type` and `f` for `feature`.) The pudding classes (including classes describing food variants such as `REPETITION`) have features such as `sugar` and `calorie_content`; the contract classes have features such as `horizon` and `value`. Two notes are in order:

- Since we started from a purely functional model, all the features mentioned so far are either creation procedures or queries. Although it is possible to keep this functional style in an object-oriented framework, the development might also introduce commands, for example, to change a contract in response to a certain event such as renegotiation. This issue—state, or not?—is largely irrelevant to the discussion of modularization.
- In the original, the `value` function yielded an infinite sequence. We can keep this signature by using a result of type `COUNTABLE`, permitting the equivalent of lazy computation; or we can give `value` an integer argument so that `value (i)` returns the `i`-th value.

The Modularization Policy

The modularization achieved so far illustrates the fundamental idea of object technology (at least the one we find fundamental [Meyer 1997]): *merging the concepts of type and module*. In its simplest expression, object-oriented analysis, design, and implementation means that we base every module of a system on a type of objects manipulated by the system. This is a more restrictive discipline than the modular facilities offered by other approaches: a module is no longer just an association of software elements—operations, types, variables—that the designer chooses to keep together based on any suitable criterion; it is the collection of properties and operations applicable to instances of a type.

The class is the result of this type-module merge. In OO languages such as Smalltalk, Eiffel, and C# (but not, for example, in C++ or Java), the merge is bidirectional: not only does a class define a type (or a type template if genericity is involved) but, the other way around, any type, including basic types such as integer, is formally defined as a class.

It is possible to retain classes in their type role only, separate from the modular structure. This is in particular the case with functional languages such as OCaml that offer both a traditional module structure and a type mechanism taken from object-oriented programming. (Haskell is similar, with a more restricted concept of class.) Conversely, it is possible to remove the requirement that all types be defined by classes, as with C++ and Java where basic types such as Int are not classes. The view of object technology taken here assumes a full merge, with the understanding that a higher-level of class grouping (packages as in Java or .NET, clusters in Eiffel) may be necessary, but as an organizational facility rather than a fundamental construct.

This approach implies the *primacy of types over functions* when it comes to defining the software architecture. Types provide the modularization criterion: every operation (function) gets attached to a class, not the other way around. Functions, however, take their revenge

through the application of abstract data type principles: a class is defined, and known to the rest of the world, through an abstract interface (API) listing the applicable operations and their formal semantic properties (contracts: preconditions, postconditions, and, for the class as a whole, invariant).

The rationale for this modularization policy is that it yields better modularity, including extendibility, reusability, and (through the use of contracts) reliability. We must, however, examine these promises concretely on the examples at hand.

Inheritance

An essential contribution of the object-oriented method to modularity goals is inheritance. As we expect the reader to be familiar with this technique, we will only recall some basic ideas and sketch their possible application to the examples.

Inheritance organizes classes in taxonomies, roughly representing the "is-a" relation, to be contrasted with the other basic relation between classes, *client*, which represents usage of a class through its API (operations, signatures, contracts). Inheritance typically does not have to observe information hiding, as this is incompatible with the "is-a" view. While some authors restrict inheritance to pure subtyping, there is in fact nothing wrong with applying it to support a standard module inclusion mechanism. Eiffel actually has a "nonconforming inheritance" mechanism (Ecma International 2006), which disallows polymorphism but retains all other properties of inheritance. This dual role of inheritance is in line with the dual role of classes as types and modules.

In both capacities, inheritance captures commonalities. Elements of a tentative taxonomy for puddings might be as described by the inheritance graph shown in Figure 13-3.

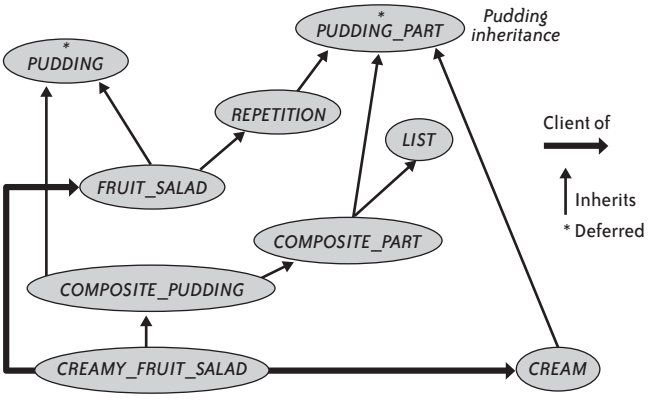

FIGURE 13-3. A class diagram of pudding ingredients

It is important to note the distribution of roles between inheritance and the client relation. A fruit salad is a pudding and is also a repetition in the earlier sense (we ignore generic parameters). A repetition is a special case not of pudding but of "pudding part," describing food ingredients. Some pudding parts (such as "composite puddings"), but not all, are also puddings. A fruit salad is a pudding and also a repetition (of fruit parts). A "creamy fruit salad," on the other hand, is *not* a fruit salad, if we take this notion to mean a pudding made of fruits only. It *has* a fruit salad and cream, as represented by the corresponding client links. It *is* a composite pudding, since this notion indeed represents concoctions that are made of several parts, like the more general notion of `COMPOSITE_PART`, and are also puddings. Here the parts, reflected in the client links, are a fruit salad and cream.

A similar approach can be applied to the contract example, based on a classification of contract types into such categories as "zero-coupon bonds," "options," and others to be obtained from careful analysis with the help of experts from that problem domain.

Multiple inheritance is essential to this object-oriented form of modeling. Note in particular the definition of a composite part, applying a common pattern for describing such composite structures (see Meyer 1997, 5.1, "Composite figures"):

```
class COMPOSITE_PART inherit
       PUDDING_PART
       LIST[PUDDING_PART]
feature
  ...
end
```

where square brackets introduce generic parameters. A composite part is both a pudding part, with all the applicable properties and operations (sugar content, etc.), and a list of pudding parts, again with all the applicable list operations: cursor movements such as `start` and `forth`, queries such as `item` and `index`, and commands to insert and remove elements. The elements of the list may be pudding parts of any of the available kinds, including—recursively—composite parts. This makes it possible to apply techniques of polymorphism and dynamic binding, as discussed next. Note the usefulness of having both genericity and inheritance; also, multiple inheritance should be the full mechanism for classes, not the form limited to interfaces (Java- and .NET-style) which would not work here.

Polymorphism, Polymorphic Containers, and Dynamic Binding

The contribution of inheritance and genericity to extendibility and extendibility comes in part from the techniques of polymorphism and dynamic binding, illustrated here by the version of `sugar_content` for class `COMPOSITE_PART` (see Figure 13-4):

```
sugar_content: REAL
      do
           from start until after loop
                Result := Result + item.sugar_content
                forth
```

```
        end
end
```

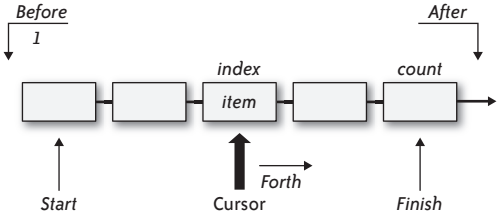

FIGURE 13-4. A polymorphic list with cursors

This applies the operations of class `LIST` directly to a `COMPOSITE_PART`, since the latter class inherits from the former. The result of `item` can be of any of the descendant types of `PUDDING`; since it may as a consequence denote objects of several types, it is known as a *polymorphic variable* (more precisely in this case, a polymorphic query). An entire `COMPOSITE_PART` structure, containing items of different types, is known as a *polymorphic container*. Polymorphic containers are made possible by the combination of polymorphism, itself resulting from inheritance, and genericity. (As these are two very different mechanisms, the functional programming term "parametric polymorphism" for genericity can cause confusion.)

The polymorphism of `item` implies that successive executions of the call `item.sugar_content` will typically apply to objects of different types; the corresponding classes may have different versions of the query `sugar_content`. *Dynamic binding* is here the guarantee that such calls will in each case apply the appropriate version, based on the type of the object actually attached to `item`. In the case of a part that is itself composite, this will be the above version, applied recursively; but it could be any other—for example, the version for `CREAM`.

Here as in most current approaches to OO design, polymorphism is controlled by the type system. The type of `item`'s value is variable, but only within descendants of `PUDDING` as specified by the generic parameter of `COMPOSITE_PART`. This is part of a development that has affected both the functional programming world and the object-oriented world: using increasingly sophisticated type systems (still based on a small number of simple concepts such as inheritance, genericity, and polymorphism) to embody a growing part of the intelligence about a system's architecture into its type structure.

Deferred Classes and Features

Classes `PUDDING` and `PUDDING_PART` are marked as "deferred" (with an asterisk in the BON object-oriented modeling notation [Walden and Nerson 1994]) in the earlier diagram of the class structure. This means they are not completely implemented; another term is "abstract class." A deferred class will generally have deferred *features*, possessing a signature and (importantly) a contract, but no implementation. Implementations appear in nondeferred ("effective")

descendant classes, adapted to the choice that each effective class has made for implementing the general concept defined by the deferred class. In the example, both classes PUDDING and PUDDING_PART have deferred features sugar_content and calories; descendants will "effect" (implement) it, for example, in COMPOSITE_PART, by defining the sugar content as the sum of the content of the parts, as shown earlier. In COMPOSITE_PUDDING, which inherits this version from COMPOSITE_PART and the deferred version from PUDDING, the effective version takes over, giving its implementation.

> NOTE
>
> **The rule is that inheriting two features with the same name causes a name clash, which must be resolved through renaming, except if one of the features is deferred and the other effective, in which case they just yield a single feature with the available implementation. It is for this kind of sound application of the inheritance mechanism that name overloading brings intractable complexity, suggesting that this mechanism should not appear in object-oriented languages.**

Deferred classes are more sophisticated than the Java and .NET notion of "interface" mentioned earlier, since they can be equipped with contracts that constrain future effectings, and also because they can contain effective features as well, offering the full spectrum between a fully deferred class, describing a pure implementation, and an effective one, defining a complete implementation. Being able to describe partial implementations is essential to the use of object-oriented techniques for architecture and design.

In the financial contract example, CONTRACT and OPTION would be natural deferred class candidates, although again they do not need to be *fully* deferred.

Assessing and Improving OO Modularity

The preceding section summarized the application of object-oriented architectural techniques to the examples at hand. We must now examine the sketched result in light of the modularity criteria stated at the beginning of this discussion. The contribution to reliability follows from the type system and contracts; we concentrate on reusability and extendibility.

Reusing Operations

One of the principal consequences of using inheritance is that common features can be moved to the highest applicable level; then descendants do not need to repeat them: they simply inherit them "as is." If they do need to change the implementation while retaining the functionality, they simply redefine (or "override") the inherited version. "Retaining the functionality" means here that, as noted, the original contracts still apply, whether the version being overridden was already effective or still deferred. This goes well with dynamic binding: a client can use the operation at the higher level—for example, my_pudding.sugar_content, or

`my_contract.value`—without knowing what version of the routine is used, in what class, and whether it is specific to that class or inherited.

Thanks to commonalities captured by inheritance, the number of feature definitions may be significantly smaller than the maximum `t` × `f`. Any reduction here is valuable: it is a general rule of software design that repetition is always potentially harmful, as it implies future trouble in configuration management, maintenance, and debugging (if a fault found its way into the original, it must also be corrected in the copies). Copy-paste, as David Parnas has noted, is the software engineer's enemy.

The actual reduction clearly depends on the quality of the inheritance structure. We note here that abstract data type principles are the appropriate guidance here: since the key to defining types for object-oriented design is to analyze the applicable operations, a properly designed inheritance hierarchy will ensure that classes that collect features applicable to many variants appear toward the top.

There seems to be no equivalent to these techniques in a functional model. With combinators, it is necessary to define the variant of every operation for every combinator, repeating any common ones.

Extendibility: Adding Types

How well does the object-oriented form of architecture support extendibility? One of the most frequent forms of extension to a system will be the addition of new types: a new kind of pudding, pudding part, or financial contract. This is where object technology shines in its full glory. Just find the place in the inheritance structure where the new variant best fits—in the sense of having the most operations in common—and write a new class that inherits some features, redefines or effects those for which it provides its own variants, and add any new features and invariant clauses applicable to the new notion.

Dynamic binding is again essential here; the benefit of the OO approach is to remove the need for client classes to perform multibranch discriminations to perform operations, as in: "if this is a fruit salad, then compute in this way, else if it is a flan, then compute in that way, else ...," which must be repeated for every operation and, worse, must be updated, for every single client and every single operation, any time a type is added or changed. Such structures, requiring client classes to maintain intricate knowledge of the variant structure of the supplier concepts on which they rely, are a prime source of architecture degradation and obsolescence in pre-OO techniques. Dynamic binding removes the issue; a client application can ask for `my_pudding.calories` or `my_contract.value` and let the built-in machinery select the appropriate version, not having to know what the variants are.

No other software architecture technique comes close to the beauty of this solution, combining the best of what the object-oriented approach has to offer.

Extendibility: Adding Operations

The argument for object technology's support for extendibility comes in part (in addition to mechanisms such as information hiding and genericity, as well as the central role of contracts) from the assumption that the most significant changes in the life of a system are of the kind just discussed: introducing a type that shares some operations with existing types and may require new operations. Experience indeed suggests that this is the most frequent source of nontrivial change in practical systems, where object-oriented techniques show their advantage over others. But what of the other case: adding operations to existing types? Some client application relying on the notion of pudding might, for example, want to determine the `cost` of making various puddings, even though pudding classes do not have a `cost` feature.

Functional programming performs neither better nor worse for the addition of an operation than for the addition of a type: it's a matter of adding 1 to `f` rather than `t`. The object-oriented solution, however, does not enjoy this neutrality. The basic solution is to add a feature at the right level of the hierarchy. But this has two potential drawbacks:

- Because inheritance is a rather strong binding ("is-a") between classes, all existing descendants are affected. In general, adding a feature to a class at a high position in the inheritance structure can be a delicate matter.
- This solution is not available if the author of the client system is not permitted to modify the original classes, or simply does not have access to their text—a frequent case in practice since these classes may have been grouped into a library, for example, a financial contract library. It would make no sense to let authors of every application using the library modify it.

Basic object-oriented techniques (e.g., Meyer 1997) do not suffice here. The standard OO solution, widely used, is the *visitor pattern* (Gamma et al. 1994). The following sketch, although not quite the standard presentation, should suffice to summarize the idea. (It is summarized from Meyer's *Touch of Class: An Introduction to Programming Well* [2008], a first-semester introductory programming textbook—suggesting how fundamental these concepts have become.) Figure 13-5 lists the actors involved in the pattern.

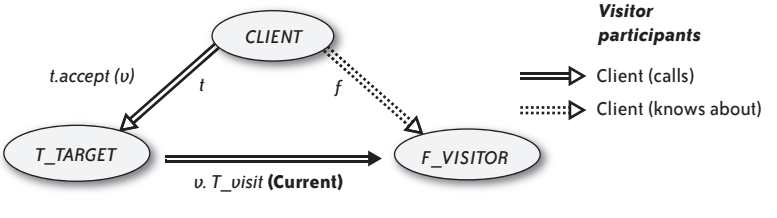

FIGURE 13-5. Actors of the Visitor pattern

The pattern turns the pas de deux between the application (classes such as CLIENT) and the existing types (such as T_TARGET for a particular type T, which could be PUDDING or CONTRACT in our examples) into a ménage à trois by introducing a visitor class F_VISITOR for every applicable operation F, for example, COST_VISITOR. Application classes such as CLIENT call an operation on the target, passing the appropriate visitor as an argument; for example:

```
my_fruit_salad.accept (cost_visitor)
```

The command accept (v: VISITOR) performs the operation by calling on its *argument* v—cost_visitor in this example—a feature such as FRUIT_SALAD_visit, whose name identifies the target type. This feature is part of the class describing such a target class, here FRUIT_SALAD; it is applied to an object of the corresponding type (here a fruit salad object), which it passes as argument to the T_visit feature. *Current* is the Eiffel notation for the current object (also known as "this" or "self"). The *target* of the call, v on the figure, identifies the operation by using an object of the corresponding visitor type, such as COST_VISITOR.

The key question in software architecture when assessing extendibility is always distribution of knowledge; a method can only achieve extendibility by limiting the amount of knowledge that modules must possess about each other (so that one can add or change modules with minimum impact on the existing structure). To understand the delicate choreography of the visitor pattern, it is useful to see what each actor needs and does not need to know:

- The target class knows about a specific type, and also (since, for example, FRUIT_SALAD inherits from COMPOSITE_PUDDING and COMPOSITE_PUDDING from PUDDING) its context in a type hierarchy. It does *not* know about new operations requested from the outside, such as obtaining the cost of making a pudding.
- The visitor class knows all about a certain operation, such as cost, and provides the appropriate variants for a range of relevant types, denoting the corresponding objects through arguments: this is where we will find routines such as fruit_salad_cost, flan_cost, tart_cost, and such.
- The client class needs to apply a given operation to objects of specified types, so it must know these types (only their existence, not their other properties) and the operation (only its existence and applicability to the given types, not the specific algorithms in each case).

Some of the needed operations, such as accept and the T_visit features, must come from ancestors. Figure 13-6 is the overall diagram showing inheritance (FRUIT_SALAD abbreviated to SALAD).

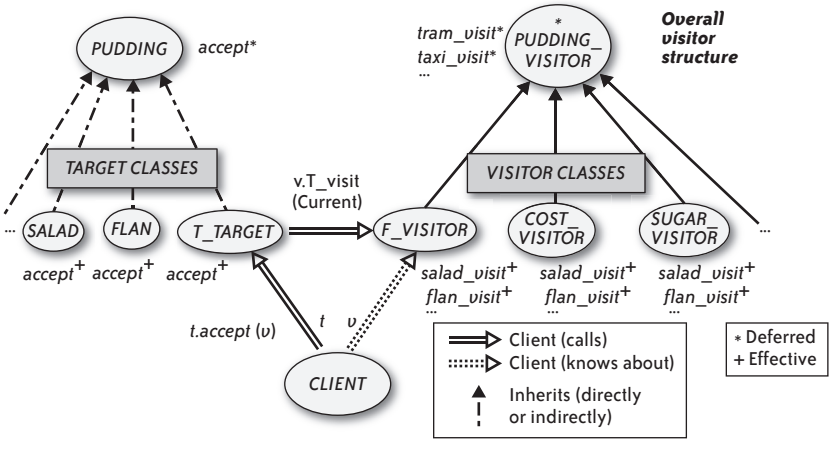

FIGURE 13-6. Putting it all together: an architecture for constructing puddings

Such an architecture is commonly used to provide new operations on an existing structure with many inheritance variants, without having to change that structure for every such operation. A common application is in language processing—for compilers and other tools in an Interactive Development Environment—where the underlying structure is an Abstract Syntax Tree (AST): it would be disastrous to have to update the AST class each time a new tool needs, for its own purposes, to perform a traversal operation on the tree, applying to each node an operation of the tool's choosing. (This is known as "visiting" the nodes, explaining the "visitor" terminology and the `T_visit` feature names.)

For this architecture to work at all, the clients must be able to perform `t.accept (v)` on any `t` of any target type. This assumes that all target types descend from a common class—here, `PUDDING`—where the feature `accept` will have to be declared, in deferred form. This is a delicate requirement since the goal of the whole exercise was precisely to avoid modifying existing target classes. Designers using the Visitor pattern generally consider the requirement to be acceptable, as it implies ensuring that the classes of interest have a common ancestor—which is often the case already if they represent variants of a common concept, such as `PUDDING` or `CONTRACT`—and adding just *one* deferred feature, `accept`, to that ancestor.

The Visitor pattern is widely used. The reader is the judge of how "beautiful" it is. In our view it is not the last word. Criticisms include:

- The need for a common ancestor with a special `accept` feature, in domain-specific classes that should not have to be encumbered with such concepts irrelevant to their application domain, whether puddings, financial contracts, or anything else.

- More worryingly, the class explosion, with numerous miniature `F_VISITOR` classes embodying a very specific kind of knowledge (a special operation on a set of special types). For the overall software architecture, this is just pollution.

Depollution requires adding a major new concept to the basic object-oriented framework: agents.

Agents: Wrapping Operations into Objects

The basic ideas of agents (added to the basic object-oriented framework of Eiffel in 1997; see also C# "delegates") can be expressed in words familiar in the functional programming literature: we treat operations (functions in functional programming, features in object-oriented programming) as "*first-class citizens*." In the OO context, the only first-class citizens are, at runtime, objects, corresponding in the static structure to classes.

The Agent Mechanism

An agent is an object representing a feature of a certain class, ready to be called. A feature call `x.f(u, ...)` is entirely defined by the feature name `f`, the target object denoted by `x`, and the arguments `u, ...`; an agent expression specifies `f`, and may specify none, some, or all of the target and arguments, said to be *closed*. Any others, not provided in the agent's definition, are *open*. The expression denotes an object; the object represents the feature with the closed arguments set to the given values. One of the operations that can be performed on the agent object is `call`, representing a call to `f`; if the agent has any open arguments, the corresponding values must be passed as arguments to `call` (for the closed arguments, the values used are those specified in the agent's definition).

The simplest example of agent expression is `agent f`. Here all the arguments are open, but the target is closed. So if `a` is this agent expression—as a result of the assignment `a := agent f`, or of a call `p (agent f)` where the formal argument of `p` is `a`—then a call `a.call ([u, v])` has the same effect as `f (u, v)`. The difference, of course, is that `f (u, v)` directly names the feature (although dynamic binding means it could be a variant of a known feature), whereas in the form with agents, `a` is just a name, which may have been obtained from another program unit. So at this point of the program, nothing is known about the feature except for its signature and, on request, its contract. Because `call` is a general-purpose library routine, it needs a single kind of argument. The solution is to use a *tuple*, here the two-element tuple `[u, v]`. In this form, `agent f`, the target is closed (it is the current object) and both arguments are open.

A variant is `agent x.f`. Here, too, the arguments are open and the target is closed: that target is `x` rather than the current object. To make the target open, use `agent {T}.f`, where `T` is the type of `x`. Then a call needs a three-argument tuple: `a.call ([x, u, v])`. To keep some arguments open, you can use the same notation, as in `agent x.f ({U}, v)` (typical call `a.call ([u])`), but since the type `U` of `u` is clear from the context, you do not need to specify it explicitly; a question

mark suffices, as in `agent x.f (?, v)`. This also indicates that the original forms with all arguments open, `agent f` and `agent x.f`, are abbreviations for `agent f (?, ?)` and `agent x.f (?, ?)`.

The `call` mechanism applies dynamic binding: the version of `f` to be applied will, as in non-agent calls, depend on the dynamic type of the target.

If `f` represents a query rather than a command, you can get from the corresponding agent the result of a call to `f` by using `item` instead of `call`, as in `a.item ([x, u, v])` (which performs a `call` and returns the value of its result); or you can call `call` and then access `a.last_result`, which, in accordance with the command-query separation principle, will return the same value, with no further `call`, in successive invocations.

For more advanced uses, rather than basing an agent on an existing feature `f`, it is also possible to write agents inline, as in `editor_window.set_mouse_enter_action (agent do text.highlight end)`, illustrating a typical use for graphical user interfaces, the basic style for event-driven programming in EiffelVision library. Inline agents provide the same mechanism as lambda expressions in functional languages: to write operations and make them directly available to the software as values to be manipulated like any other "first-class citizens."

More generally, agents enable the object-oriented framework to define higher-level functionals just as in functional languages, with the same power of expression.

Scope of Agents

Agents have turned out to be an essential and natural complement to the basic object-oriented mechanisms. They are widely used in particular for:

- Iteration: applying a variable operation, naturally represented as an agent, to all elements in a container structure.
- GUI programming, as just noted.
- Mathematical computations, as in the example of integrating a certain function, represented by an agent, over a certain interval.
- Reflection, where an agent provides properties of features (not just the ability to call them through `call` and `item`) and, beyond them, classes.

Agents have proved essential to our investigation of how to replace design patterns by reusable components (Arnout 2004; Arnout and Meyer 2006; Meyer 2004; Meyer and Arnout 2006). The incentive is that while the designer of any application needing a pattern must learn it in detail—including architecture and implementation—and build it from scratch into the application, a reusable component can be used directly through its API. Success stories include the Observer design pattern (Meyer 2004; Meyer 2008), which no one having seen the agent-based solution will ever be tempted to use again, Factory (Arnout and Meyer 2006), and Visitor, as will be discussed next.

An Agent-Based Library to Make the Visitor Pattern Unnecessary

The agent mechanism permits a much better solution to the problem addressed somewhat clumsily by the Visitor pattern: adding operations to existing types, without changing the supporting classes. The solution is detailed in Meyer and Arnout (2006) and available through an open source library available on the download site of the ETH Chair of Software Engineering (ETH Zurich, Chair of Software Engineering, at *http://se.ethz.ch*).

The resulting client interface is particularly simple. No change is necessary to the target classes (`PUDDING`, `CONTRACT`, and such): there is no more `accept` feature. One can reuse the classes exactly as they are, and accept their successive versions: there is no more explosion of visitor classes, but a single `VISITOR` library class, with only two features to learn for basic usage, `register` and `visit`. The client designer does not need to understand the internals of that class or to worry about implementing the Visitor pattern, but only needs to apply the basic scheme for using the API:

1. Declare a variable representing a visitor object, specifying the top target type through the generic parameter of `VISITOR`, and create the corresponding object:

```
pudding_visitor: VISITOR [PUDDING]
create pudding_visitor
```

2. For every operation to be executed on objects of a specific type in the target structure, register the corresponding agent with the visitor:

```
pudding_visitor.register (agent fruit_salad_cost)
```

3. To perform the operation on a particular object—typically as part of a traversal—simply use the feature `visit` from the library class `VISITOR`, as in:

```
pudding_visitor.visit (my_pudding)
```

That is all there is to the interface: a single visitor object, registration of applicable operations, and a single visit operation. Three properties explain this simplicity:

- The operations to be applied, such as `fruit_salad_cost`, would have to be written regardless of the architecture choice. Often they will already be available as routines, making the notation `agent fruit_salad_cost` possible; if not—especially if they are very simple operations—the client can avoid introducing a routine by using inline agents. In either case, there is no need for the spurious `T_visit` routines.
- It seems strange at first that a single `VISITOR` class, with a single `register` routine to add a visitor, should suffice. In the Visitor pattern solution the calls `t.accept (v)`, the target `t` identified the target type (a particular kind of pudding), but here `register` does not specify any such information. How can the mechanism find the right operation variant to apply (the cost of a fruit salad, the cost of a flan)? The answer is a consequence of the reflective properties of the agent mechanism: an agent object embodies all the information about the associated feature, including its signature. So `agent fruit_salad_cost` includes the

information that this is a routine applicable to fruit salads (from the signature `fruit_salad_cost (fs: FRUIT_SALAD)`, also available, in the case of an inline agent, from its text). This makes it possible to organize the internal data structures of `VISITOR` so that in a visiting call, such as `pudding_visitor.visit (my_pudding)`, the routine `visit` will find the right routine or routines to apply based on the dynamic type of the target, here `pudding_visitor: VISITOR [P]` for a specific pudding type `P`—also matching, as enforced statically by the type system, the type of the object dynamically associated with the argument, here the polymorphic `my_pudding`.

- This technique also enjoys the reuse benefits of inheritance and dynamic binding: if a routine is registered for a general pudding type (say, `COMPOSITE_PUDDING`) and no other has been registered for a more specific type (for example, the cost might be computed in the same way for all composite puddings), `visit` uses the best match.

The mechanism as described provides the complement to traditional OO techniques. When the problem is to add types providing variants of existing operations, inheritance and dynamic binding work like a charm. For the dual problem of adding operations to existing types without modifying these types, the solution described here will apply.

Applying the previous modularity criterion of distribution of knowledge—*who must know what?*—we see that in this approach:

- Target classes only know about fundamental operations, such as `sugar_content`, characterizing the corresponding types.
- An application only needs to know the interface of the target classes it uses, and the two essential features, `register` and `visit`, of the `VISITOR` library class. If it needs new operations on the target types, not foreseen in the design of the target classes, such as `cost` in our example, it need only provide the operation variants that it needs for the target types of interest, with the understanding that in the absence of overriding registration, the more general operations will be used for more specific types.
- The library class `VISITOR` does not know anything about specific target types or specific applications.

It seems impossible to go any further in minimizing the amount of knowledge required of the various parts of the system. The only question that remains open, in our opinion, is whether such a fundamental mechanism should remain available through a library or should somehow yield a language construct.

Assessment

The introduction of agents originally raised the concern that they might cause redundancy and hence confusion by offering alternative solutions in cases also amenable to standard OO mechanisms. (Such concerns are particularly strong in Eiffel, whose language design follows the principle of providing "*one good way to do anything.*") This has not happened: agents

found right away their proper place in the object-oriented arsenal; designers have no trouble deciding when they are applicable and when not.

In practice, all nontrivial uses of agents—in particular, the cited pattern replacements—also rely on genericity, inheritance, polymorphism, dynamic binding, and other advanced OO mechanisms. This reinforces the conviction that the mechanism is a necessary component of successful object technology.

> NOTE
>
> **For a differing opinion, see the Sun white paper explaining why Java does not need an agent- or delegate-like facility (Sun Microsystems 1997). It shows how to emulate the mechanism using Java's "inner classes." Although interesting and well-argued, it mostly succeeds, in our view, at demonstrating the contrary of its thesis. Inner classes do manage to do the job, but one can readily see, as in the elimination of the Visitor pattern with its proliferation of puny classes, the improvement in simplicity, elegance, and modularity brought by an agent-based solution.**

Agents, it was noted above, allow object-oriented design to provide the same expressive power of functional programming through a general mechanism for defining higher-order functionals (operations that can use operations—themselves recursively enjoying the same property—as their inputs and outputs). Even lambda expressions find their counterpart in inline agents. These mechanisms were openly influenced by functional programming and should in principle attract the enthusiasm of its proponents, although one fears that some will view this debt acknowledgment as an homage that vice pays to virtue (La Rochefoucauld 1665).

Setting aside issues of syntax, the only major difference is that agents can wrap not only pure functions (queries without side effects) but commands. Ensuring full purity does not, however, seem particularly relevant to discussions of architecture, at least as long as we enforce the command-query separation principle, retaining the principal practical benefit of purity—referential transparency of expressions—without forcing a stateful model into the artificial stranglehold of stateless models.

Agents bring the final touch to object technology's contribution to modularity, but they are only one of its elements, together with those sketched in this discussion and a few more. The combination of these elements, going beyond what the functional approach can offer, makes object-oriented design the best available approach to ensure beautiful architecture.

Acknowledgments

I am grateful to several people who made important comments on drafts of this contribution; obviously, acknowledgment implies no hint of endorsement (as is obvious in the case of the functional programming grandees who were kind enough to share constructive reactions without, I fear, being entirely swayed yet by my argument). Particularly relevant were observations by Simon Peyton Jones, Erik Meijer, and Diomidis Spinellis. John Hughes's

answer to my questions about his classic paper were detailed and illuminating. The Visitor library discussed in the last part of this chapter is the work of Karine Arnout (Karine Bezault). I thank Gloria Müller for further observations as part of an ETH master's thesis on implementing a library of Haskell-like facilities for Eiffel. I am especially grateful to the editors of this volume, Diomidis Spinellis and Georgios Gousios, for the opportunity to publish this discussion and for their extreme patience with my delays in finalizing it.

References

Arnout, Karine. 2004. "From patterns to components." Ph.D. thesis, ETH Zurich. Available at *http://se.inf.ethz.ch/people/arnout/patterns/*.

Arnout, Karine, and Bertrand Meyer. 2006. "Pattern componentization: the Factory example." *Innovations in Systems and Software Technology* (a NASA Journal). New York, NY: Springer-Verlag. Available at *http://www.springerlink.com/content/am08351v30460827/*.

Eber, Jean-Marc, based on joint theoretical work with Simon Peyton Jones and Pierre Weis. 2001. "Compositional description, valuation, and management of financial contracts: the MLFi language." Presentation available at *http://www.lexifi.com/Downloads/MLFiPresentation.ppt*.

Ecma International. 2006. *Eiffel: Analysis, Design and Programming Language*. ECMA-367. Available at *http://www.ecma-international.org/publications/standards/Ecma-367.htm*.

Frankau, Simon, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. 2008. "Commercial uses: Going functional on exotic trades." *Journal of Functional Programming*, 19(1):2745, October.

Gamma, Erich, et al. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

Hughes, John. 1989. "Why functional programming matters." *Computer Journal*, vol. 32, no. 2: 98–107 (revision of a 1984 paper). Available at *http://www.cs.chalmers.se/~rjmh/Papers/whyfp.pdf*.

La Rochefoucauld, François de. 1665. *Réflexions ou sentences et maximes morales*.

Meyer, Bertrand, and Karine Arnout. 2006. "Componentization: the Visitor example." *Computer* (IEEE), vol. 39, no. 7: 23–30. Available at *http://se.ethz.ch/~meyer/publications/computer/visitor.pdf*.

Meyer, Bertrand. 1992. *Eiffel: The Language*. (Second printing.) Upper Saddle River, NJ: Prentice Hall.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*, Second Edition. Upper Saddle River, NJ: Prentice Hall. Available at *http://archive.eiffel.com/doc/oosc/*.

Meyer, Bertrand. 2004. "The power of abstraction, reuse and simplicity: An object-oriented library for event-driven design." *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*. Eds. Olaf Owe, Stein Krogdahl, and Tom Lyche. *Lecture Notes in Computer Science*, 2635, pp. 236–271. New York, NY: Springer-Verlag. Available at *http://se.ethz.ch/~meyer/publications/lncs/events.pdf*.

Meyer, Bertrand. 2008. *Touch of Class: An Introduction to Programming Well*. New York, NY: Springer-Verlag. See *http://touch.ethz.ch*.

Peyton Jones, Simon, Jean-Marc Eber, and Julian Seward. 2000. "Composing contracts: An adventure in financial engineering." Functional pearl, in *ACM SIGPLAN International Conference on Functional Programming* (ICFP '00), Montreal, Canada, September '00. ACM Press, pp. 280–292. Available at *http://citeseer.ist.psu.edu/jones00composing.html*.

Peyton Jones, Simon, and Philip Wadler. 1993. "Imperative functional programming." *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Charleston, South Carolina, pp. 71–84. Available at *http://citeseer.ist.psu.edu/peytonjones93imperative.html*.

Steingold, Sam. Online at *http://www.podval.org/~sds/ocaml-sucks.html*.

Sun Microsystems. 1997. "About Microsoft's 'Delegates.'" White paper by the Java Language Team at JavaSoft. Available at *http://java.sun.com/docs/white/delegates.html*.

Wadler, Philip. 1995. "Monads for functional programming." *Advanced Functional Programming*, Lecture Notes in Computer Science 925. Eds. J. Jeuring and E. Meijer. New York, NY: Springer-Verlag. Available at *http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf*.

Walden, Kim, and Jean-Marc Nerson. 1994. *Seamless Object-Oriented Software Architecture*. Upper Saddle River, NJ: Prentice Hall. Available at *http://www.bon-method.com/index_normal.htm*.