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