- 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