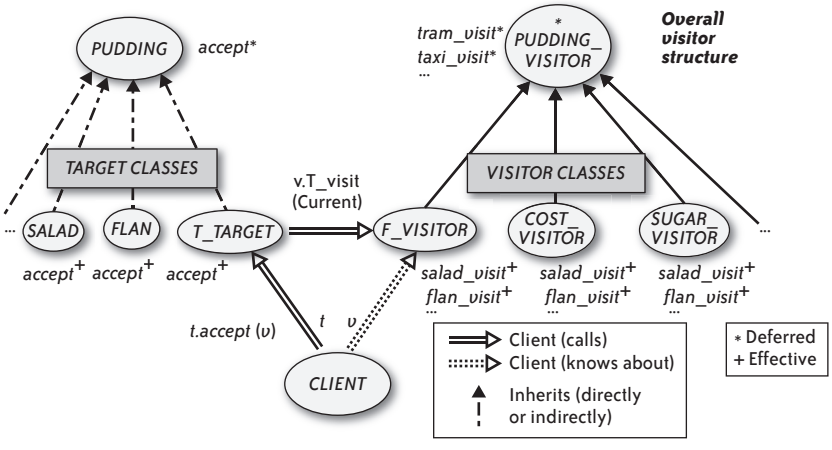

FIGURE 13-6. Putting it all together: an architecture for constructing puddings

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.