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).