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.

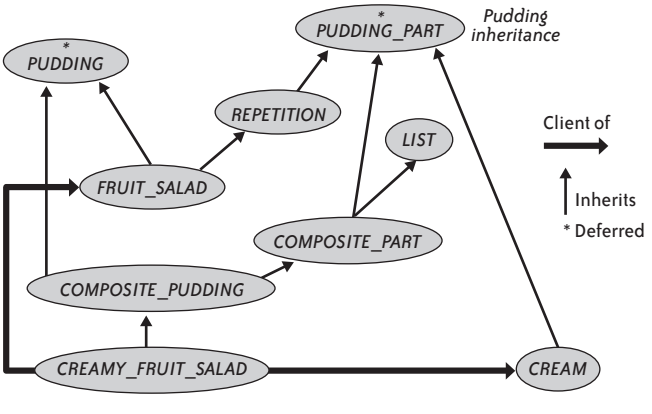

FIGURE 13-3. A class diagram of pudding ingredients