Not only can classes have different creation procedures, they will generally have many more features. Specifically, the *operations* of our previous versions become features of the appropriate classes. (The reader may now have guessed that the variable name `t` stood for `type` and `f` for `feature`.) The pudding classes (including classes describing food variants such as `REPETITION`) have features such as `sugar` and `calorie_content`; the contract classes have features such as `horizon` and `value`. Two notes are in order:

- Since we started from a purely functional model, all the features mentioned so far are either creation procedures or queries. Although it is possible to keep this functional style in an object-oriented framework, the development might also introduce commands, for example, to change a contract in response to a certain event such as renegotiation. This issue—state, or not?—is largely irrelevant to the discussion of modularization.
- In the original, the `value` function yielded an infinite sequence. We can keep this signature by using a result of type `COUNTABLE`, permitting the equivalent of lazy computation; or we can give `value` an integer argument so that `value (i)` returns the `i`-th value.

The Modularization Policy

The modularization achieved so far illustrates the fundamental idea of object technology (at least the one we find fundamental [Meyer 1997]): *merging the concepts of type and module*. In its simplest expression, object-oriented analysis, design, and implementation means that we base every module of a system on a type of objects manipulated by the system. This is a more restrictive discipline than the modular facilities offered by other approaches: a module is no longer just an association of software elements—operations, types, variables—that the designer chooses to keep together based on any suitable criterion; it is the collection of properties and operations applicable to instances of a type.

The class is the result of this type-module merge. In OO languages such as Smalltalk, Eiffel, and C# (but not, for example, in C++ or Java), the merge is bidirectional: not only does a class define a type (or a type template if genericity is involved) but, the other way around, any type, including basic types such as integer, is formally defined as a class.

It is possible to retain classes in their type role only, separate from the modular structure. This is in particular the case with functional languages such as OCaml that offer both a traditional module structure and a type mechanism taken from object-oriented programming. (Haskell is similar, with a more restricted concept of class.) Conversely, it is possible to remove the requirement that all types be defined by classes, as with C++ and Java where basic types such as Int are not classes. The view of object technology taken here assumes a full merge, with the understanding that a higher-level of class grouping (packages as in Java or .NET, clusters in Eiffel) may be necessary, but as an organizational facility rather than a fundamental construct.

This approach implies the *primacy of types over functions* when it comes to defining the software architecture. Types provide the modularization criterion: every operation (function) gets attached to a class, not the other way around. Functions, however, take their revenge