It is important to note the distribution of roles between inheritance and the client relation. A fruit salad is a pudding and is also a repetition in the earlier sense (we ignore generic parameters). A repetition is a special case not of pudding but of "pudding part," describing food ingredients. Some pudding parts (such as "composite puddings"), but not all, are also puddings. A fruit salad is a pudding and also a repetition (of fruit parts). A "creamy fruit salad," on the other hand, is *not* a fruit salad, if we take this notion to mean a pudding made of fruits only. It *has* a fruit salad and cream, as represented by the corresponding client links. It *is* a composite pudding, since this notion indeed represents concoctions that are made of several parts, like the more general notion of `COMPOSITE_PART`, and are also puddings. Here the parts, reflected in the client links, are a fruit salad and cream.

A similar approach can be applied to the contract example, based on a classification of contract types into such categories as "zero-coupon bonds," "options," and others to be obtained from careful analysis with the help of experts from that problem domain.

Multiple inheritance is essential to this object-oriented form of modeling. Note in particular the definition of a composite part, applying a common pattern for describing such composite structures (see Meyer 1997, 5.1, "Composite figures"):

```
class COMPOSITE_PART inherit
       PUDDING_PART
       LIST[PUDDING_PART]
feature
  ...
end
```

where square brackets introduce generic parameters. A composite part is both a pudding part, with all the applicable properties and operations (sugar content, etc.), and a list of pudding parts, again with all the applicable list operations: cursor movements such as `start` and `forth`, queries such as `item` and `index`, and commands to insert and remove elements. The elements of the list may be pudding parts of any of the available kinds, including—recursively—composite parts. This makes it possible to apply techniques of polymorphism and dynamic binding, as discussed next. Note the usefulness of having both genericity and inheritance; also, multiple inheritance should be the full mechanism for classes, not the form limited to interfaces (Java- and .NET-style) which would not work here.

Polymorphism, Polymorphic Containers, and Dynamic Binding

The contribution of inheritance and genericity to extendibility and extendibility comes in part from the techniques of polymorphism and dynamic binding, illustrated here by the version of `sugar_content` for class `COMPOSITE_PART` (see Figure 13-4):

```
sugar_content: REAL
      do
           from start until after loop
                Result := Result + item.sugar_content
                forth
```