descendant classes, adapted to the choice that each effective class has made for implementing the general concept defined by the deferred class. In the example, both classes PUDDING and PUDDING_PART have deferred features sugar_content and calories; descendants will "effect" (implement) it, for example, in COMPOSITE_PART, by defining the sugar content as the sum of the content of the parts, as shown earlier. In COMPOSITE_PUDDING, which inherits this version from COMPOSITE_PART and the deferred version from PUDDING, the effective version takes over, giving its implementation.

> NOTE
>
> **The rule is that inheriting two features with the same name causes a name clash, which must be resolved through renaming, except if one of the features is deferred and the other effective, in which case they just yield a single feature with the available implementation. It is for this kind of sound application of the inheritance mechanism that name overloading brings intractable complexity, suggesting that this mechanism should not appear in object-oriented languages.**

Deferred classes are more sophisticated than the Java and .NET notion of "interface" mentioned earlier, since they can be equipped with contracts that constrain future effectings, and also because they can contain effective features as well, offering the full spectrum between a fully deferred class, describing a pure implementation, and an effective one, defining a complete implementation. Being able to describe partial implementations is essential to the use of object-oriented techniques for architecture and design.

In the financial contract example, CONTRACT and OPTION would be natural deferred class candidates, although again they do not need to be *fully* deferred.

Assessing and Improving OO Modularity

The preceding section summarized the application of object-oriented architectural techniques to the examples at hand. We must now examine the sketched result in light of the modularity criteria stated at the beginning of this discussion. The contribution to reliability follows from the type system and contracts; we concentrate on reusability and extendibility.

Reusing Operations

One of the principal consequences of using inheritance is that common features can be moved to the highest applicable level; then descendants do not need to repeat them: they simply inherit them "as is." If they do need to change the implementation while retaining the functionality, they simply redefine (or "override") the inherited version. "Retaining the functionality" means here that, as noted, the original contracts still apply, whether the version being overridden was already effective or still deferred. This goes well with dynamic binding: a client can use the operation at the higher level—for example, my_pudding.sugar_content, or