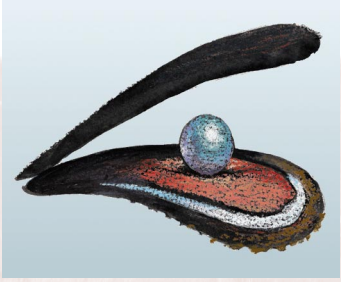
Tell Less, Say More: The Power of Implicitness

Bertrand Meyer, EiffelSoft

When we are asked to describe something, our natural impulse is to reach for an exhaustive definition. If what we are asked to describe is a software concept—and if we are using a programming language—we will usually start writing a record, structure, or class definition, and we won't come back until we have all the properties in place. A bank account is the combination of the account holder's name, a current balance amount, an account number, an allowed minimum balance, and so on; a paragraph (in a text processing system) is the combination of a list of words, a default font, a justification rule, and so on. But we won't feel comfortable until we have removed every single "and so on" and filled in all the details.

How does object technology affect this? The first part of the answer is well known. It's that the properties used in the description are no longer confined to attributes or data members, indicating what physically constitutes an object: Now they also include routines or methods, indicating applicable operations. So we will not just have a record or structure declaration, which in a Pascal-like language would read

Editor: Bertrand Meyer, EiffelSoft, ISE Bldg., 2nd Fl., 270 Storke Rd., Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com



The open-closed principle is one of the central innovations of object technology.

```
type ACCOUNT =
   record
      holder: PERSON;
      ... Other attribute
declarations ...
   end
```

but will obtain a true class by adding the routines, as in the following Eiffel form:

```
class ACCOUNT feature
   holder: PERSON

   change_holder
(new_holder: PERSON) is
      do .. end

   ... Other attribute and
routine declarations
end
```

This example illustrates the principal and well-known difference, which also defines why a C++ class is not the same thing as a C structure.

But there is another difference, no less important even if it is less obvious. The first form is explicit; the second is implicit.

OPEN-CLOSED PRINCIPLE

Implicitness here means that we do not claim to have listed all the properties—or care. In any class declaration there remains a possible "and so on." We accept that the description may be incomplete, yet we can use it as it is.

This is the *open-closed principle*, which in my opinion is one of the central innovations of object technology: the ability to use a software component as it is, while retaining the possibility of adding to it later through inheritance. Unlike the records or structures of other approaches, a class of object technology is both closed *and* open: closed because we can start using it for other components (its clients); open because we can at any time add new properties without invalidating its existing clients.

The open-closed principle helps address a particularly pressing problem of software engineering: smooth evolution. In the situation illustrated in Figure 1, we have a successful module A serving the needs of some clients. When new clients such as B1 appear with their own needs, non-OO approaches might force us either to change the original, thereby running the risk of invalidating all the existing clients (such as B and C), or to write a new module, causing duplication of software and probably duplication of bugs and future maintenance efforts.

In the OO approach, we avoid this dilemma by writing A so that it is directly usable by B and other clients, yet is still open for adaptation through inheritance. A1, a descendant of A, does not duplicate the properties of A that are still applicable in the context of A1; it only defines what is different or new in that context.

SAYING ENOUGH

Applying the open-closed principle as a way of life in OO development means

that the "and so on" of our lists of properties is always there, even if not stated explicitly. We do not need or want to be exhaustive.

If we later need to refine the concepts of interest, we can do so without invalidating the existing descriptions and—most importantly, from the viewpoint of large-scale and long-term software engineering—their clients. This does not mean, of course, that we should abuse inheritance and (like some object-oriented designers I have seen) introduce a new class each time we need a new feature. A class should always define a coherent and significant abstraction, and if we are just adding a property that we initially forgot, we should simply update the class definition. But for major adaptations and extensions, we can afford to get the new and keep the old.

This absence of any claim to exhaustiveness gives a Zen-like quality to the process of OO modeling and design. We do not pretend that we try to say everything of interest, or even that we could. We say what we know of the objects of interest. We say what objects *have*; we don't pretend to have defined what they *are*. As we learn more, we say more. And at some stage we'll just feel we have enough—not all, just enough. At least for the moment.

THE FINITE-FUNCTION MODELS

Looking at mathematical models can help you understand what's going on. (I know that software people don't "talk math" in polite company. But there is really no reason why in a scientific discipline we couldn't summon the help of math to gain some insight into our problem domain, especially when the math involved is elementary.)

If we stick for simplicity to classes that only have attributes—the equivalent of records—the most tempting model for something like

```
class ACCOUNT feature
    account_number: INTEGER
    current_balance: REAL
end
```

is a Cartesian product: We can think of this declaration as representing the set INTEGER × REAL—that is to say, the set of pairs <*i*, *r*>, where *i* is an INTEGER and *r* is a REAL. Although not wrong in any formal sense, this Cartesian product model creates a paradox: We cannot subclass and subset at the same time!

Under no circumstance can A × B × C be mapped onto a subset of A × B; it's actually the other way around. So if we are considering adding to ACCOUNT a new attribute, say address: STRING, we cannot consider the result a subset of the original, even though that's what we want since an "account with number, balance, and address" is a special case of an "account with number and balance." Object technology allows us—through polymorphism and dynamic binding—to treat an instance of the latter as if it were an instance of the former.

To remove the Cartesian product paradox, it suffices to use a different model, one based on partial functions. Consider a set T of tags that includes all tags of interest, such as *account_number*, *current_balance*, *address*, and a set V of values, which includes all values of interest, such as integers, reals, and strings. Then we can consider a class such as ACCOUNT as describing partial functions from tags to values with the property that every such function must be defined for a finite set of values including both the tags *account_number* and *current_balance*. Also, the value for *account_number* must be an integer and the value for *current_balance* must be a real number.

AN EXAMPLE

An example is {<*account_number*, 9087123>, <*current_balance*, 499.95>}, the finite function defined for exactly the two tags given, and describing a particular object—a particular bank account with the two attributes valued as shown (account number 9087123 and current balance $499.95).

Then an instance of ACCOUNT_WITH_ADDRESS is a finite function defined for the tags *account_number*, *current_balance*, and address. By its very construction, this is also an instance of ACCOUNT as defined.

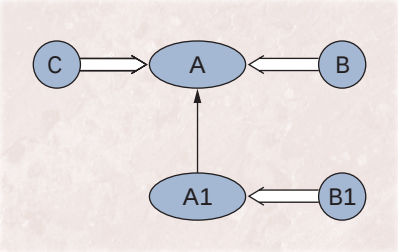


Figure 1. Reconciling openness and closure through inheritance.

In the Zen spirit of implicitness, we never exactly specify the domains of our functions (the precise set of tags for which they are defined); we simply list tags that these domains must include. We do not say that the domain of ACCOUNT must *be* the set {*account_number*, *current_balance*}; we simply require that it *include* the two tags given.

This is what makes it possible to consider descendant classes as subsets, even though they have more features. Like its software counterparts, this mathematical model is both open and closed.

Under these mathematical observations lies a very practical feature of OO development, as distinctive as anything that is routinely considered part of the definition of the approach—information hiding, classes, genericity, inheritance, polymorphism, dynamic binding, contracts. It's the constant refusal to say more than what we strictly need to say.

The double refusal—refusal to close, refusal to assert completeness—requires some intellectual audacity, but yields a productive development process. In the end, what resolves the contradiction is the release of the software. The release process—the process of closing what was until now open—is the process of equating each type with the Cartesian product of its properties. So when everything has been said, the objects *are* indeed what they have. But only at the end. Until then, you always leave room for more properties.

Everything is open until officially closed; and by telling less now you retain the possibility of saying more later. ❖