Attributes? Why?

William Kent
Database Technology Department
Hewlett-Packard Laboratories
Palo Alto, California

July 1992

Why is there sky?

Things in object models should be there for useful reasons.

Objects exist in computational systems which enable users to do things. The users might be interactive people or programmed applications.

Users need objects and operations because that's how they get things done: by applying operations to objects. Users need specifications because they need to know what various operations will do when applied to various objects so they can pick the right ones to accomplish their goals.

Users need types so that specifications don't have to be tediously repeated for individual objects - especially objects that don't even exist yet. Instead of saying that the XYZ Manual can be edited and printed and the memo I wrote you yesterday can be edited and printed and the letter you will write tomorrow morning can be edited and printed and so on and on, object specifications simply say that documents can be edited and printed. Then anything which is classified as a document can be edited and printed.

Users need subtyping and inheritance so that specifications can be shared and reused via refinement and redefinition. We might have to say that source programs can be compiled, but if we say they are documents then we don't have to repeat that they can also be edited and printed.

What do users need to know about state? In a vague sort of way, they need some assurance that enough will be remembered to capture the effect of one operation on another. If the user requests that an object be moved, he expects that to be remembered the next time he asks where the object is, or how far it is from some other object. If the user pushes and pops things on a stack, he expects enough to be remembered so that the next pop yields the right thing.

What can users know about state? In one sense, state is the sum total of all information that could be returned by operations at a given moment -- a hypothetical snapshot as though all operations could be executed independently at the same time. But even that's not enough. There is often no operation which reveals

a snapshot of the state of a stack, which can then only be revealed by an indefinite sequence of Pops.

But even if it was possible to reveal a snapshot of state by executing all operations, that seems redundant. What seems more relevant is some independent subset of that information, sufficient to allow all the rest to be deduced. If there were operations that revealed the radius, diameter, circumference, and area of a circle, we wouldn't need all of these in a minimal subset. Knowing one of these is sufficient to figure them all out. But which one? There's another rub. Any one of them will do. If we want to think of state as the minimum information that needs to be remembered, that does not define a unique set. How shall we decide which of those four properties constitutes the state of the circle? We should certainly allow different implementations to choose different ones, and perhaps more than one for efficient retrieval.

Users don't need specific knowledge of state when they request an object to be moved or copied. They want enough information to be moved or copied so that all the operations will work in the new location. Users don't care how that is implemented.

Users don't really care if the state of one object can be isolated from the state of another. Does the state of an object include how far it is from various other objects? What about the minimum distance to any object? Those certainly are part of the information available about the object. Does an object's state change when some other object moves? What difference does it make how we answer those questions? (If your answer is that such things are relationships rather than attributes, then we unleash the same sorts of questions about relationships, compounded by the problem of precisely distinguishing between attributes and relationships.)

Users do very much care about specifications of what operations will do. A user certainly wants to know what operation he should request so that the next time he asks for Sam's age the answer will be 40. But that's no different from the user's need to know the effects of any operation. Users need to know the effects of Hire, Fire, Rotate, Zoom, Crop, Pan, Promote, Position, Price, Push, Pop, Print, Shred, etc. etc. Object specifications currently do a notoriously poor job of explaining this to the user. All we tell him officially is the type of object to which he can apply the operation, and the types of results he might expect. All the rest he has to glean from the name of the operation, how it is implemented (there might be several implementations) and, with luck, some comments - with no assurance from the model that anything he guesses about the behavior is correct.

For some obscure reason we single out certain update operations for special attention, explaining their behavior more fully via the attribute construct. Proposals typically confound semantics and syntax, often indistinguishable because the specifications provide little specification beyond examples.

The essential semantics associate an updating operation with another operation whose results are being modified. Thus for a certain operation foo there may be an associated updating operation bar such that bar(x,y) causes foo(x) to return y. In effect there is a conceptual association such that Updater(foo)=bar. For example, we might have Updater(Locate) = Move. Not all operations have updaters. Some updaters might be implemented by simply changing stored data, while others may perform computations, usually culminating in a change to stored data. Thus the updater for Diameter might simply store the new value, or it might store half of it as the radius.

Most proposals for attributes illustrate a particular syntax: declaring an attribute A implicitly defines a pair of operations Get_A and Set_A such that Updater(Get_A)=Set_A. The proposals never say whether the illustrative naming convention is a necessary part of the specification, or why. The naming convention does not seem to be needed by any facility within the object system, such as binding, routing, or dispatching. Such syntactic naming conventions may be appropriate for a specific language binding, but seem inappropriate in semantic model specifications. It's not obvious that such naming restrictions are always welcome to the user. Does every operation beginning with Set_ have to be an updating operation? Is there any good reason why the user can't say Age(Sam) rather than Get_Age(Sam)? Is there any good reason why the updater for Locate can't be Move?

There are many things in life whose popularity I can't fathom, from presidents to pop rock to pet rocks to attributes. I'm sorely tempted to write off the persistence of attributes as nothing more than an unshakable bad habit, a hangover, a vestigial relic of prior technologies. Historically, attributes were associated with stored data. We all accept that this is no longer appropriate in object specifications. What's left? Am I really missing something big here?

Attributes can't capture all of state, as exemplified by stacks.

State information is not uniquely determined, as with the radius, diameter, circumference, and area of a circle. How does the designer know which things to designate as attributes?

The state of an object may not be separable from the states of other objects, as with the distances to other objects.

The functionality of attributes is redundant with operations. The reason for making the distinction, and the consequences, must confuse the user.

Non-updatable attributes mystify me altogether.

Proposals confound semantics and syntax, introducing arguable naming conventions.

Behavior of other operations is not adequately described. Why single out update?

Stored data, and what has to be copied or moved with an object, is certainly of very serious concern to implementers. Maybe this concern is being unjustifiably projected onto the user. Compilers may be able to take advantage of implementation specifications. Let's not confuse that with semantic interface specifications for users. Perhaps the implications of a clean separation between semantic interface specifications and implementation concerns needs to be more carefully thought through.

There may also be an implicit worry about inefficiency stemming from an unstated presumption that an operation is necessarily implemented as executable code, rather than having the implementation specified as a mapping to stored data. Perhaps the underlying assumption is that operations must be implemented as code, whereas attributes can be implemented either as code or as storage mappings. There's no reason why an implementation of an operation couldn't also be specified as a mapping to stored data. The default behavior for a corresponding updater would then simply update the same stored data.

The ability to specify a corresponding updater for a given operation seems semantically useful. Other than that, do we need attributes?

Why?

http://www.bkent.net/Doc/whyatt.htm