User Object Models

William Kent
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303-0971, USA
kent@hplabs.hp.com

Abstract

Object users and developers have different needs, contributing to the plethora of object models. This paper focuses on user object models, differentiating them from developer models, and outlines a spectrum of characteristics which can provide a basis for comparing and reconciling different user models.

Keywords: object orientation, object models.

Contents

1 Introduction

This is yet another assault on the elusive object model. Maybe part of the problem is that it means different things to object users and object developers.

The object paradigm provides an encapsulation boundary shielding object users (people or programs) from object implementations. Object users can request a controlled set of operations on a controlled set of operands at the encapsulating interface.

Object developers define the operations and the kinds of operands to which they may be applied, together with their implementations in code and data structures. Such code and data constitutes another set of operations and operands at a different interface, generally considered to be at a lower level of abstraction, used to implement the higher-level interface. A given object request interface can be implemented in many ways by different mappings to lower interfaces ([HZ]). Application code accrues the benefits of object orientation — reusability, sharability, interoperability — to the extent that it adheres to a higher-level interface, being usable on a variety of different implementations.

Object system developers may have yet another perspective, being concerned with the infrastructure that manages the invocation, communication, and execution of requests.

These people may all have different things in mind when they think about objects. Metaphorically speaking, a user looks down at an interface from above, while a developer looks up at an interface from below. The same person/code could be an object developer with respect to an interface "above" and an object user with respect to an interface "below".

Object models differ in the extent to which they differentiate between user and developer views, and the extent to which they focus on one or the other. They differ in the essential nature of objects as presented to users, and in the degree to which developer's implementation concerns are encapsulated from users.

This paper differentiates between user and developer views of objects, focusing primarily on the user view. Even within the user view, there is a spectrum of concepts regarding what an object is. We present a spectrum of characteristics, as a framework for describing and comparing object models [K1]. We don't try to define or characterize the entire O-O paradigm, focusing only on the core concept of what an object *is*. We'd like to put that into perspective before getting to secondary issues like what we mean by types and classes and inheritance and polymorphism and identity and so on.

Our result is not a single model, but a multi-dimensional space of possibilities in which various models can be positioned, a checklist of questions and criteria for identifying and articulating the differences among models. It sets the stage and defines the playing field, but doesn't solve the problem of reconciling diverse object models.

2 An Operational Framework

The crux of the object paradigm, spanning user and developer views, centers on the roles that objects play with respect to requested operations. Figure 1 illustrates a user's request to start keeping a certain product in a certain warehouse; in response, he is told the particular bin allocated to that product.

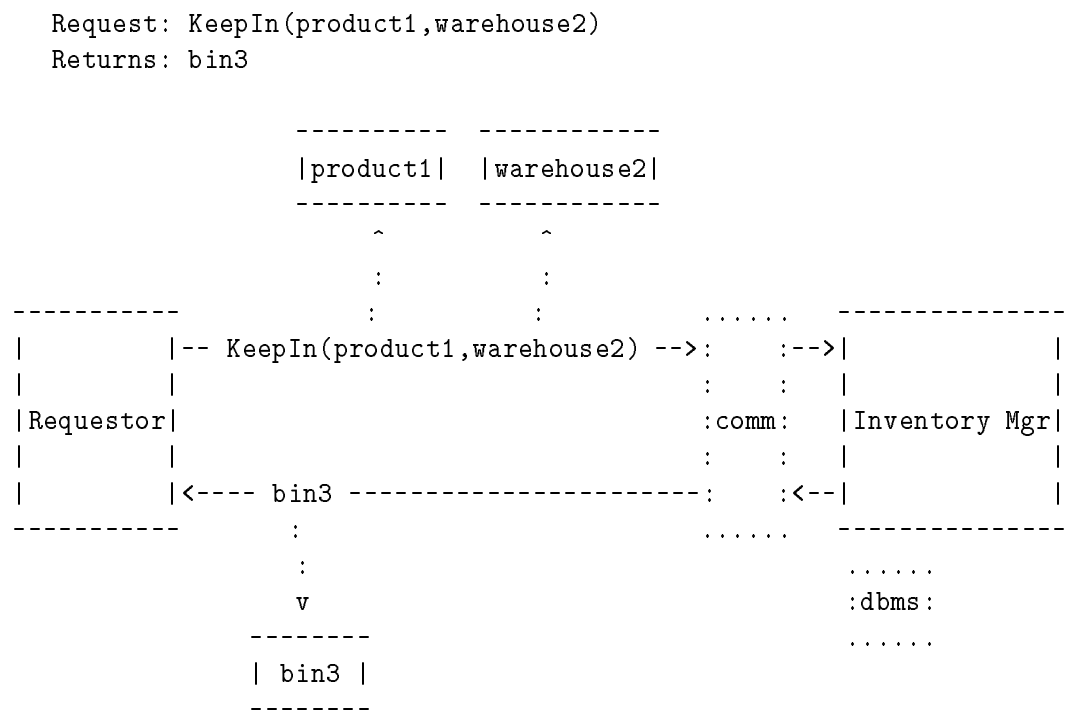

Figure 1: The operational paradigm.

Various things may be involved in this operational paradigm:

- A requestor.
- A request: KeepIn(product1,warehouse2).
- An operator: KeepIn.
- Operands: product1,warehouse2.
- Results: bin3.
- A service provider: the inventory manager application, and its components.
- Other system facilities: communication channels and services, dbms, etc.
- Auxiliary data constructs: links, tables, relationships, etc.
- The context in which the request is made.
- The *kinds* of things involved, as distinguished from the individual things themselves: Product, Warehouse, Bin, Operator, etc.

Are these all objects? If they are, then the most general notion of *object* is something which plays a role with respect to a requested service. Models differ as to which roles are recognized, and which of those are considered relevant to characterizing an object.

What the user is most obviously aware of are the operator, its operands, and the results.

3 What Does the User See?

The object user sees:

- An object request interface.
- Consequences of making requests at that interface.
- Specifications describing the consequences he should expect.

Consequences of making a request include:

- *Applicability*, i.e., whether it is even legal to request that operation with those operands. If not, the consequence is some sort of error, e.g., an undefined operation or operand, or a type violation.
- *Results*, i.e., things returned as a "value" of the operation.
- *Side effects*, manifested to the user by the altered consequences of future requests.

We will, in the course of this paper, also characterize things the user shouldn't see, being more appropriate to the developer's view. For instance, the user doesn't really know to which object a request "goes"; the notion may not even have any meaning for the user. Whether the request "goes to" a product or a warehouse, or to an inventory application, or to a dbms; whether it cascades through intermediate services; and whether it involves an "inventory table" as a data object, are things that might concern the developer, but not the requestor.

4 The Object Request Interface

The very first question concerns how the user perceives the object request interface in relation to the rest of his environment. The user may or may not have to distinguish object requests from other requests. In the course of his business, the requestor might do arithmetic and string operations, call subroutines, make procedure calls, make requests of operating systems, of data bases, and of other system components, and do other similar things. Do object requests include some or all of these activities, or are they a distinct kind of activity in their own right?

Suppose Contents(Bin) and Price(Product) are object requests. If the object request interface is seamlessly integrated with the programming environment, then the user could request a composed operation like Price(Contents(bin3)), or evaluate an expression like Price(product1)

+ Price(product2). Otherwise the user has to do such things piecemeal, dealing separately with independent interfaces.

If an object model differentiates object requests from other requests, then there must be some convention of syntax or context by which the user establishes the distinction. One should then explain the initial *triage* mechanism which differentiates the various sorts of requests. It may or may not be an identifiable facility; it might simply be implicit in a variety of coding and interface conventions, comparable to the way a compiler recognizes the difference between an arithmetic operation and a subroutine call.

Models might be differentiable by the semantic criteria by which they distinguish object requests from other requests, and the syntactic conventions by which the distinction is made manifest.

5 Description and Classification

The roles that objects can play with respect to requests could be described for individual objects. In most cases, though, it's more practical to describe that generically for a group of objects. In any case, since one of the actions users can request is the creation of new objects, it makes sense to describe the kinds of objects that can be created at an interface. In order to understand the descriptions, the user needs to know how objects are classified.

We have a chicken-and-egg situation: does classification determine characteristics, or vice versa? Can an object be printed because it is a document, or is it a document because it can be printed?

In some models, objects acquire characteristics in an unspecified way. They then have to meet certain criteria to be considered a certain kind of object; e.g., an object has to support printing in order to be a document. In other models, creating and manipulating an object with legal operations guarantee that it will have certain characteristics. Controlled creation and editing of a document guarantee it can be printed.

There are many potential criteria for classifying things in the object paradigm. Of particular interest from the user viewpoint are the roles things can play as operands or results of operations. In the spirit of [OM, OO], we will use *type* to refer to this classification mechanism.

6 Describing Operations

The following things are relevant to the user's understanding of an operation:

- The types of things to which it may be applied as operands.
- The types of things returned as results, if any.
- The things on which it has "side effects", if any.
- Explanations of these consequences.

The following developer concerns should not concern the user:

- The actual code implementing the operation, including its organization and grouping with other units of code, or which object "contains" the code.
- Whether the operation is supported by a distinct copy of the code for each object, or whether one copy of the code is shared by objects of the same kind.
- The data structures and other mechanisms used to manage state and side effects.

6.1 Operands and Results: Generalized Role Model

For operations which take single simple operands, it suffices to say whether or not the operation is applicable to a type of object. For operations which take multiple or complex operands, we should be more specific.

Consider an operation with signature of the form

Foo(Person, Set of <Person, City>) → Integer.

It's not enough to say that Foo is applicable to a person, or to a city. We have to say that a person may occur as the first operand, but not as the second. We don't want to say that a city can be the second operand, but rather that it can occur as a certain part of a structure that can occur as the second operand. Similar things may need to be said about complex results.

We can introduce the notion of a *signature role* as a labeled part of the operand and result structure:

Foo(r1 Person, r2 Set of <r3 Person, r4 City>) → r5 Integer.

For the operation Foo, we can say that a person can play roles r1 or r3, a city can play role r4, a certain kind of set can play role r2, and an integer can play the role r5 in the result. Roles thus provide the linkage between operations and object types. An object type is associated with a set of roles; objects of that type can play those roles.

Notationally, when an operation takes multiple operands without complex structure, we could simply refer to the roles positionally, e.g., as the first or second operand within a given operation. We will use this simpler form in examples.

In general, the relationship of operators to types of objects can be specified in operator signatures (Figure 2).

Figure 2: Operator signatures.

6.2 Side Effects

Side effects are discussed in Section 8. Minimum information required here is an indication of which requests, if any, are affected by this operation. Current object technology does not generally have effective means for describing the side effects of operations to the user, independently of implementations.

6.3 Explanations

The reader may wonder what some of the operations in Figure 2 really do. All we see there are the operand and result types. We didn't say anything about the algorithm that determines the specific results returned, or about side effects of the operation.

Well, that's all the explanation an object user typically gets in most object models, unless he looks at the method code and data structures which implement the operation. Ideally, such things would be described in powerful, user-friendly specification languages defining an understandable and enforceable contract between users and developers. They would be described only in terms of other constructs exposed to the user, and not any underlying implementations.

It would be desirable to tell the user, for example, that QOH(Product,Whse) returns the quantity on hand of a given product in a given warehouse, as the summation of QOH(Product,Bin) for that product in bins in that warehouse. That tells the user what the operation means, i.e., why a particular integer value is returned. It also tells the user about side effects: altering QOH(Product,Bin) affects the result returned by QOH(Product,Whse).

The user should be able to see this without knowing whether the value is maintained in storage or computed on demand, or whether it is computed by an iterative procedure or by a set-oriented database query. Similarly, the user does not care whether updating

QOH(Product,Bin) requires code to be executed that alters a cached value of QOH(Product,Whse). Those are developer's implementation concerns; such decisions should be made independently, and be changeable, without altering the user's model.

Unfortunately, the full capability seems to be beyond the current state of the art. The closest we can come today are:

- Readable comments, having no influence on correct implementation.
- High-level languages which are readable, but are incomplete in expressive power or for which there is not a sufficient range of efficient compilers or interpreters for all the desired implementation environments.

This is a problem (opportunity?) in current object technology, representing a significant leak in the encapsulation boundary. The only way to explain important things to the user about the behavior of an operation is by exposing the implementation, i.e., the developer's view. The boundary between the user view and the developer view is rather flimsy in this respect.

7 Describing Objects

As its very name suggests, the object-oriented paradigm is organized around object-centered descriptions, rather than the operator-centered descriptions shown above. Descriptions are clustered around object types in various ways, giving rise to various notions of what objects "are" with respect to "containing" or "holding" operations.

7.1 Signature Roles

The simplest way to cluster by object type is to group together all operations having a given type of object anywhere in their signature (Figure 3). An operation having several types of objects in its signature would be included with each of those types.

Descriptions in this form suggest that users think of objects as things which can occur as operands or results of operations. As our example suggests, it is not customary to create such clusters for the literal types, e.g., Integer and Boolean.

7.2 Operand Roles: Generalized Applicability

Operations can be clustered by applicability, i.e., by the types of objects which may occur as their operands (Figure 4). In such a *generalized* model [OM], an operation may be *jointly held* by the types of objects which may occur as its operands. Operations are not included in the descriptions of objects which only occur as results. (We are not trying to be specific about the notion of an object "holding" an operation. It is about the same idea as an operation being in an object interface [OM].)

This form of description suggests that users think of objects as things to which operations are applied.

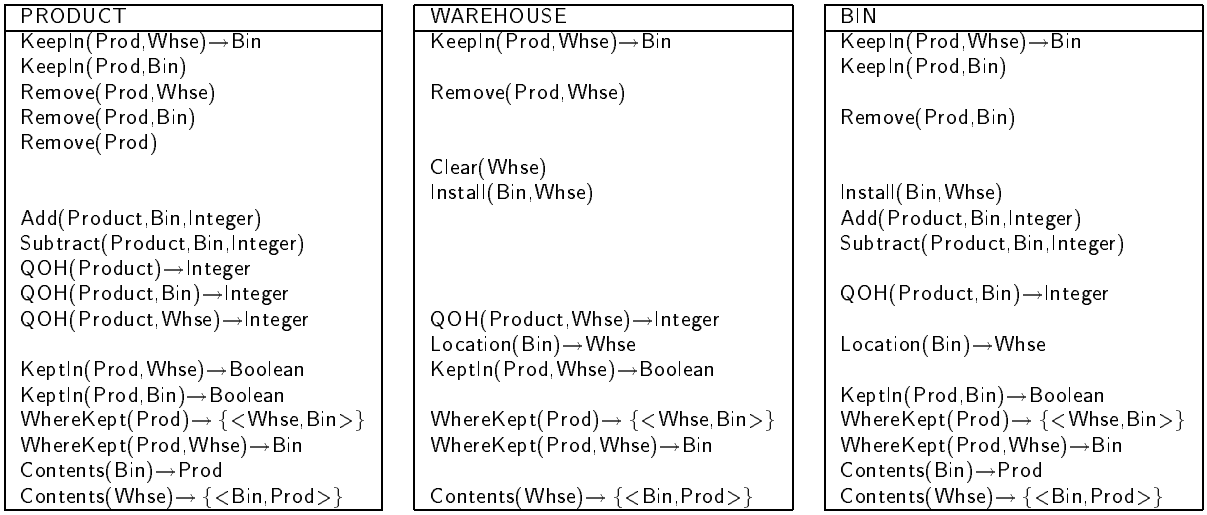

Figure 3: Clustering by signature roles.

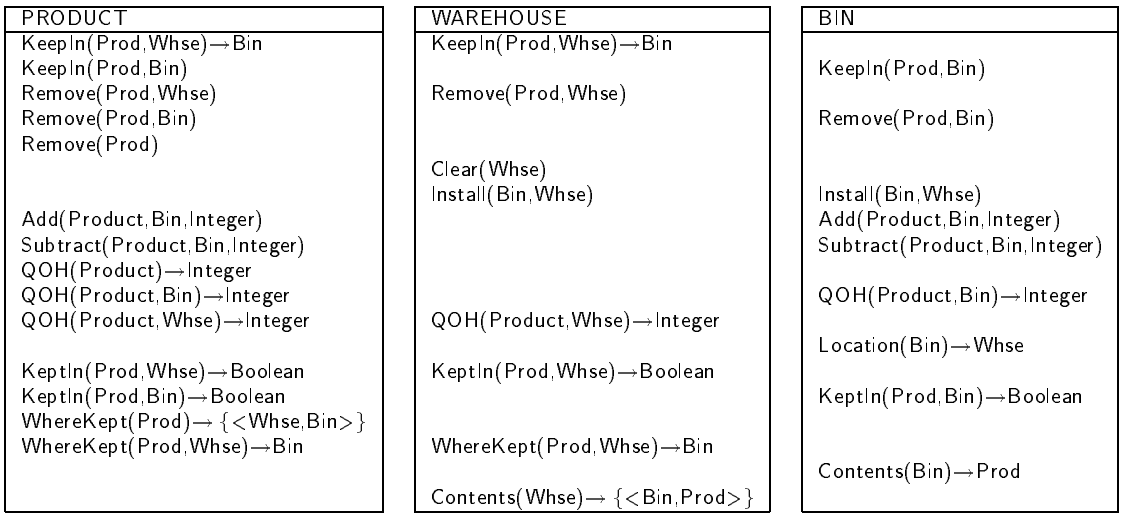

Figure 4: Clustering by operand roles.

Note that omission of result roles renders incomplete the description of what information is available for a given type of object, particularly if query capability is available. One thing we might learn about a warehouse is the set of bins located there, i.e., the bins b such that $Location(b)=w$. The availability of that information is not part of the description of warehouses in Figure 4.

7.3 Recipient Roles: Classical Messaging

Classical object models [OM] do not support joint holding of operations. One of the operand roles is distinguished as a "recipient", which exclusively holds the operation. The operation is characterized as a "message" sent to that recipient, and the other operands are considered "parameters" of the message.

Whether a given type of object may occur as a parameter to an operation is not included with the description of that type of object, but must be found in the description of the recipient.

This form of description suggests that users think of objects as things to which messages are sent. The choice as to which operand is considered the recipient is generally quite arbitrary. It is potentially another leakage of the developer's implementation decisions, if it dictates anything about how the implementing code or data structures are organized. Figure 5 shows the form of object descriptions if we arbitrarily choose the first operand to be the recipient.

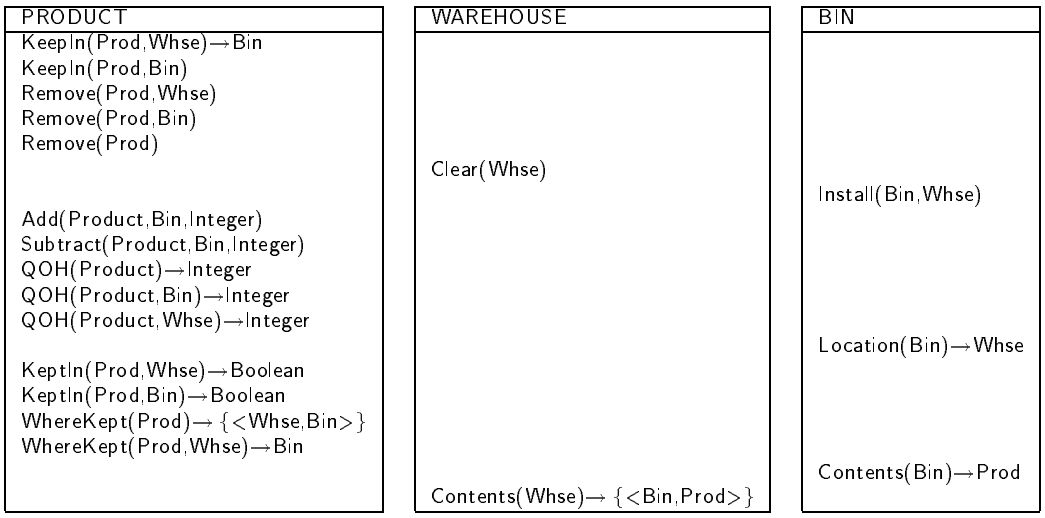

Figure 5: Clustering by recipient roles.

In such models, the request syntax often separates the recipient from the parameters, so that one might write product1.KeepIn(warehouse2) instead of KeepIn(product1,warehouse2), suggesting that the message is "sent to" the product, with the warehouse passed as a parameter. If the warehouse had been designated the recipient, the user would then have to write warehouse2.KeepIn(product1) instead.

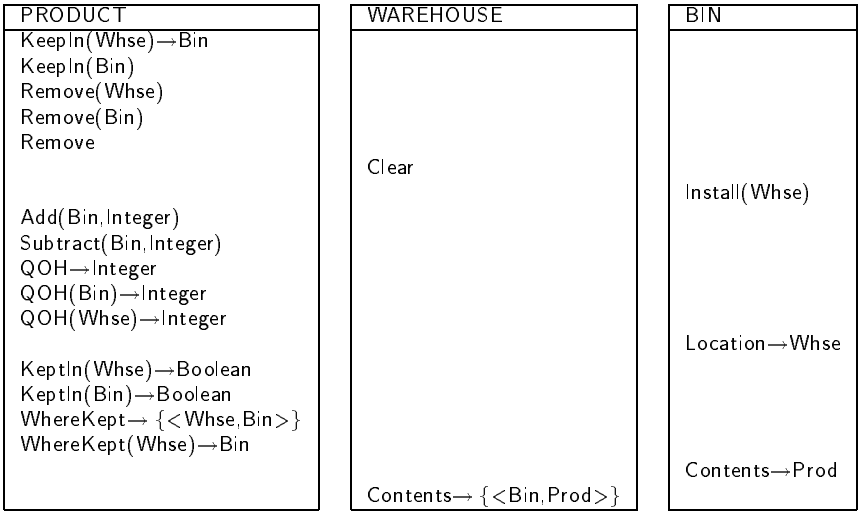

Figure 6: Compact object descriptions.

Figure 6 shows semantic specifications written more compactly, factoring out the common recipient. Only the parameter and result types are shown.

7.4 The Significance

There's more involved than just user convenience and readability of clustered specifications. The set of operations which can be offered to the user is affected.

In the classical messaging model, which is the most common, an object (more precisely, an object type) is the exclusive holder of a set of operations — and it cannot hold two operations with the same name. Different types can hold operations with the same name.

Our example turns out to be invalid in most messaging models. By choosing Product to be the recipient of both KeepIn(Product,Whse) and KeepIn(Product,Bin), we made Product the holder of two operations named "KeepIn" — which most messaging models don't allow. On the other hand, if we had arbitrarily reversed our decision and made Warehouse and Bin the recipients, with Product passed as a parameter, it would have been perfectly legal.

Clustering might also have implications regarding the "extent" (Section 9.2) of an object with respect to operations which display, move, or copy an object.

8 State

The long-term consequences of a requested operation, i.e., its side effects, are manifested in the consequences of other requests. Creating and destroying objects impacts the applicability of operations to such objects. A KeepIn request alters the results returned by KeptIn and WhereKept. Adding or subtracting quantities of products alters the results returned by the QOH operations. Pushing or popping things on a stack alters the results of the next pop.

The notion of state is an essential aspect of the object paradigm. From the user viewpoint, it can be characterized most generally as whatever is needed to realize the effect of one request on the consequences of another. There is clearly some sort of memory involved, since time can elapse between the affecting and affected requests. Users certainly need to understand such consequences. Object models differ substantially in how the notion of state is manifested, and the extent to which the manifestation blurs the encapsulation boundary between user and developer.

State might be described to the user in terms of:

1. Arbitrary operations, like KeepIn, Remove, Add, and Subtract, with explanations of their effect on other requests (Section 6.3).
2. "Assignable" (updatable) operations, for which operations of the form

   Assign(QOH,product1,100)

   are legal, equivalent to

   QOH(product1)←100.
3. Attributes (variables) included in object descriptions, with the following implications:
   (a) They are different from operations.
   (b) There is different syntax for retrieval.
   (c) Implied existence of corresponding operations for retrieval and assignment.
   (d) Values are stored, not computed.
   (e) Attributes contain the state of objects.

Item 3(d) reflects a softening of the encapsulation barrier, exposing the developer's implementation commitments to the user. There are numerous cases in which the decision to store or compute is arbitrary (birthday vs. age; radius, diameter, circumference of a circle; QOH in a warehouse as stored or summed over bins). Conversely, update of stored values sometimes needs to be intercepted by code to do validation or to propagate consequences. Whether or not such things are considered part of the "state" of objects seems to be an arbitrary decision, and should not be dependent on whether they are implemented as stored data.

Item 2 can be implemented in code that takes the appropriate action on assignment, effectively a second "update" entry point in the method [K3, KL].

What a user needs to know about an operation are:

- What it returns, if anything.
- What side effects it has, if any.
- Whether it is assignable.

In a generalized model, those could be orthogonal, and independent of whether the operation is implemented as code, stored data, or both. Those are developer concerns.

The distinction between operations and attributes is not essential, especially if the "degenerate" implementation of an operation simply maps it to stored data. Some models perceive objects as being things which have behavior and state as two distinguishable aspects. If necessary, one could distinguish a certain set of operations as being attributes, and/or as reflecting state.

The state of an object may or may not be revealed by any single operation or request. For example, the state of a stack might only be revealed by a sequence of pop operations.

As with operations, there is again the possibility of joint or exclusive holding. The state which is altered by the KeepIn operation is simultaneously the state of a product, a warehouse, and a bin. Assigning that state to be exclusively held by one of those three objects would be a highly arbitrary decision. Ideally, such a decision should have no effect on the behavior of user requests, and should not imply any implementation commitments. Models which require such decisions encourage users to think of objects as things which contain state data, rather than allowing state to be shared among objects.

9 Special Operations

Models reflect what they think objects are by the operations they presume apply to all objects.

9.1 Placement

Some objects have a sense of place within a computer system, which could be modeled to the user as a Place property that can be retrieved or altered. Such a sense of place is reflected in operations that move an object, or try to determine where an object is in order to send a message to it.

(We don't define "place" any further; we mean whatever concept is associated with the notion of moving an object, or of routing a message to where it is.)

At higher levels of abstraction, a user may not have a sense of place for all objects. It may not appear to him that a warehouse or a product is at any particular place in the computer system.

It is certainly plausible that some objects have a sense of place. Some models rest on a very deeply ingrained assumption that all objects have a place which can be identified or changed. In such models, an object is something which is at a particular place.

9.2 Extent

Users may have various "views" [HZ] of the information associated with an object. It may or may not include information associated with related objects. It may include information returned by all operations applicable to the object, or only that information considered to be the state of the object, if that is defined in the model (Section 8). Some models make

different assumptions as to what information is intrinsic to an object; the text might be considered an inherent part of a document, while information about authors, publishers, and royalty payments are not. The user may only be interested in some subset of any of this information for his purposes.

In general, operations such as display, move, or copy might be invoked with a view parameter specifying the information to be involved.

It is certainly plausible that some objects have a defined extent. Some models assume that such operations are applicable to all objects, without a view parameter, thereby assuming that an object is something which has an inherently defined extent of associated information — which may or may not coincide with assumptions about the state of the object.

Models supporting a notion of "deep equality" make the same assumption, namely that there is a well-defined notion of object content which can be compared between two objects.

9.3 Coherence

There are various sorts of coherence an object might exhibit:

- The operations held by an object are disjoint from the operations held by other objects, i.e., operations are not shared. (More properly said about object types than instances.)
- The operations held by an object (type) are all implemented at the same place.
- The operations held by an object (type) are all implemented in the same unit of code, e.g., they are all in one application.
- The state associated with an object is disjoint from the state associated with other objects, i.e., no shared state.
- The state associated with an object is all implemented in one place.
- The state associated with an object is implemented in a contiguous chunk of storage.

Models differ in the extent to which they assume that an object is something which has such coherence. It is not clear whether such assumptions matter in the user model. They seem largely relevant to the developer's view.

9.4 Object Creation

9.4.1 Types as Objects

All object models support an object creation operator. What is its operand? The object about to be created doesn't exist yet. What usually occurs as the operand is the type of object being created. Is the type an object? Some models say yes, types are objects, being instances of the type Type, which is an instance of itself. Object creation can then be described with a signature like other operators:

Create: Type → Object.

Other models do not consider types to be objects. Object creation then has to be described in some special way. That's worth comparing in different models.

9.4.2 Literals as Objects

Literal values, such as numbers and strings, always exist; they can't be created or destroyed [K2]. If a model assumes that objects are things that users can create and destroy, then literal values are disqualified as objects.

In such models, one needs to introduce other terminology for the union of objects and literal values, in order to talk about common characteristics, e.g.,

- They can all occur as operands or results of operations.
- They are organized in types, including subtype relationships.
- Developers can define subtypes, as in enumerated subtypes.
- Developers can define new operations on them.
- Such operations can be overloaded.

The notion of state can't be used to distinguish between objects and literals, since null state is usually admissible. At best, one might say that all literals have null state, which still allows them to be objects.

For some reason, object models do not consider literal types as defining interfaces, i.e., their operations do not need to be held by types.

9.5 Operators as Objects

We said at the beginning that one of the roles a thing might play in a request is the operation itself. Some models don't consider operations to be objects.

If we did, then we would have Operator as a type of object. The ability to apply an operator to operands can be abstractly modeled by an Apply operation which takes an operator as its first operand.

The assignability of operations (Section 8) would be modeled by an Assign operator taking an operator as its first operand.

Other relevant operations on operations would include creation and destruction, as well as definition or alteration of signatures and other specifications.

10 The Developer View

This is just a brief summary, largely to contrast what shouldn't be in the user view.

An object developer defines the object types and operations which may be used at a given level of abstraction, and specifies their implementation, usually in terms of constructs at a lower level of abstraction. A system developer defines an infrastructure which manages the invocation and communication of requests and their results.

Some user and developer activity could go on at the same interface. Defining types and operations could be permitted in the same interface as creating instances and requesting operations. Their behavior could be specified in terms of objects and operations existing at the same level. In general, though, developer activity spans different levels of abstraction.

Developers see a lot of objects that users don't, at a level of abstraction below the level being implemented. In a developer's view, a request might go to an object not even mentioned in the request.

Object developers are aware of the specific code and data structures implementing objects and operations. They know whether QOH(Product,Warehouse) is kept as stored data or computed on demand from the quantities in the bins at the warehouse. They know whether it is coded as an iterative procedure or a set-oriented database query. They know whether it exists as an independent code module, or is packaged with other code in an inventory management application. They know whether all operations applicable to products and warehouses are implemented in that application.

Developers know whether KeptIn(Product,Warehouse) is maintained as a list of warehouses within a product data structure, as a list of products within a warehouse data structure, as an independent data table, or by some other means.

Developers know whether the same code and data structures are used for all products and warehouses, or whether different implementations are used for different instances. Instances of the Product or Warehouse types might be subdivided into *classes* of instances sharing the same implementation (again following the nomenclature of [OM, OO]).

The encapsulation boundary is blurred to the extent that users have to choose the implementations of objects they create, or the extent to which they are in any way aware of the classes to which objects belong. Such dependencies on implementation diminish the benefits of the object-oriented paradigm, reducing the interoperability, sharability, and reusability of applications. When a user creates an instance of a type, the choice of implementations should be implicit, perhaps based on the context of the creation request (e.g., the locally installed class libraries, the nature of the machine or system environment, the organizational affiliations of the requestor), or perhaps even dynamically tuned according to the pattern of requests made on the object.

Descriptions of object behavior at the type level should be independent of descriptions at the class level. Ideally, class specifications would be constrained to conform to type specifications.

Developers know and care about mechanisms for reusing implementations.

The system developer thinks about sending a request someplace, and returning the results someplace. He knows about intermediate facilities which may be involved, such as com-

munication channels and services, or database management systems. He thinks primarily of chunks of code which have to be found, activated, and coordinated, which use data resources, have requests routed to them, issue other requests in turn, have results returned to them, might fail in various ways, and so on. He thinks in terms of coherent, placed objects to which requests and results are routed.

Users generally have it easier if dispersed (non-coherent) objects are supported. Developers have it easier with coherent objects. Dispersed objects are likely to map to coherent objects at lower levels of abstraction.

Who translates user-model requests into developer-model requests? Who figures out the operators and objects in the developer request? Making the user model conform to the developer model, by assuming all objects are fully coherent and placed, is one way to evade the problem.

At lower levels of abstraction, the developer's view may rest on other fundamental metaphors, such as an object being something which sends and receives messages, or a self-contained chunk of code and data. The perception of an application as an object, as a source and destination for messages, is more the developer's perspective than the user's. Executable objects correspond to a developer's notion of reusable code modules.

11 Conclusions

We have tried to characterize the distinction between user and developer views of the object paradigm, as perhaps one of the reasons we seem to have so many different object models. Even within the user view, though, there are many perceptions of what an object *is* — even before we deal with secondary characteristics like polymorphism and inheritance.

We take an operational paradigm as a framework for comparing user object models, and raise a number of questions within that framework:

- Does the object model differentiate user and developer views? Is it more concerned with one or the other?
- Does the model provide an effective encapsulation boundary?
- Are object requests integrated with the rest of the user's environment? What are the semantic criteria and syntactic conventions for differentiating object requests from other activities?
- Are object characteristics the cause or effect of classification? Can the user independently create and "import" things to the object system, so that they have to meet certain criteria to be of certain types? Or is he limited to creating and manipulating objects of specified types using applicable operations, thereby insuring the objects have the appropriate characteristics?
- Does the model provide a way to describe the behavior of an operation (algorithm, side effects) independently of its implementation?
- Does the model support operations having multiple operands, or complex type structures in their signatures?

- What is the most general concept of "object" in the model?
  - An object can play any role with respect to a request.
  - An object can occur as the operand or result of a request.
  - An object is something to which an operator can be applied (i.e., the object can occur as an operand).
  - An object is the recipient of a request, which is considered a message to the object.
  - Something else.
- Is everything satisfying that criterion considered an object? If not, why not?
- Can operations be jointly held by multiple objects (i.e., be in the interfaces of different types of objects), or must they be exclusively held?
- Are objects considered to have visible attributes, apart from operations?
- Does the specification of attributes imply implementation commitments regarding data storage and code?
- Is object state characterized in terms of attributes or operations?
- Can state be jointly held by multiple objects, or must it be exclusively held?
- Is it assumed that each object is at a single place in the computer system? Is a Move operation applicable to all objects?
- Is it assumed that each object has a well defined extent of associated information? How is that established? Is a Display operation applicable to all objects? Are Move and Copy operations, or a deep equality comparison, applicable to all objects?
- Are types objects? Are literals? Are operators?
- Are classes (implementation specifications) exposed to users?

Such questions can provide a basis for comparing user object models, or for arriving at a unifying model. A corresponding investigation of developer models would be useful.

12 Acknowledgments

This work evolved with the help of colleagues in the ANSI SPARC Object Oriented Database Task Group [OO] and the OMG Object Model Task Force [OM].

13 References

[HZ] Sandra Heiler and Stanley Zdonik, "Object Views: Extending the Vision", Proc. Sixth Intl Conf on Data Engineering, Los Angeles, Feb. 1990.

[K1] William Kent, "A Framework for Object Concepts", HPL-90-30, Hewlett-Packard Laboratories, April 1990.

[K2] William Kent, "A Rigorous Model of Object Reference, Identity, and Existence", Journal of Object-Oriented Programming (to appear mid-1991). HPL-90-31, Hewlett-Packard Laboratories, April 1990.

[K3] William Kent, "Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language", HPL-91-25 Hewlett-Packard Laboratories, March 1991.

[KL] Ravi Krishnamurthy, Witold Litwin and William Kent, "Language Features for Interoperability of Databases with Schematic Discrepancies", Proc ACM SIGMOD Int'l Conf on Mgmt of Data, Denver, Colorado, May 29-31 1991. Also HPL-DTD-90-14, Hewlett-Packard Laboratories, Dec. 17, 1990.

[OO] Technical Report of the ANSI/X3/SPARC/DBSSG Object-Oriented Database Task Group, (in preparation).

[OM] *Object Management Architecture Guide,* Object Management Group, Framingham MA, 1990.