The Object-Oriented Tutorial Series

To ODB or Not To ODB, That Is the Question
Part I

By Craig S. Mullins

So what's all the fuss about object orientation anyway? It seems like all we ever hear about anymore is the inevitable paradigm shift from procedural systems to object-oriented systems. That was OK when they were just talking about programming, but now it seems to be everywhere. And, even more importantly, speaking as a data technology professional, they're starting to put an "O" on my data base. I've seen both ODB (object data base) and OODBMS (object-oriented data base management system) bandied about in the industry trade journals. What do I need to know? Just what does object-oriented (OO) mean? How will it affect my DBMS-based development? Is OO just another buzzword that I can safely ignore until the next one comes buzzing along?

Well, I can sure sympathize with those of you who have these types of questions about object orientation. Unfortunately (or maybe fortunately after reading this article), you will not be able to ignore OO for too much longer. OO technology provides the system development life cycle with too many benefits to simply fade away. Yes, it seems as if OO technology will be with us for some time to come. Therefore, the time has come to begin understanding all of the concepts that are implied by the term object-oriented.

As data analysts and data base professionals, OO can provide many great benefits to the fields of data administration and data base design. Because of these benefits, OO technology is almost a shoo-in to be incorporated into business DBMS implementations in the not too distant future. Although object-oriented data base management systems currently exist, few if any, have been implemented within the large corporate accounts that relational DBMS systems dominate. But DBMS technology is not alone in being impacted by OO technology; the disciplines that surround the data technology professional have also been impacted. OO modeling, OO design, OO analysis, and evidently, anything else you can stick two "O"s in front of, will be impacted by the impending OO revolution. It is probably a wise course of action, then, for data technology professionals to become familiar with OO terminology and functionality.

But will it be a revolution or an evolution that brings OO techniques into our data bases? Is it possible or realistic to expect DP professionals and the businesses they support to embrace (both technically and financially) another DBMS revolution? In the mid '80s DB2 was at the forefront of the relational revolution. Industry pundits and academia predicted, correctly in this instance, that future DBMS development would be founded upon the relational model instead of the then popular network (IDMS) and hierarchic (IMS) models. Now DB2 owns the industry for new mainframe-based, corporate application development. The revolution occurred, but it did not completely topple the legacy systems that were based on earlier DBMS technology. For example, few companies completely replaced all of their IMS-based systems with DB2. Most, however, have embraced DB2 as the DBMS platform of the present and foreseeable future.

What does this bode for ODBMS? Two routes exist:

- revolution, completely jettisoning the relational model for an OO model and replacing DB2 with an ODBMS based upon that model; or
- evolution, adoption of key OO concepts into the relational model followed by adapting DB2 to incorporate OO functionality and techniques.

Which is more realistic? Which will provide a better development environment? Which will finally be chosen? We can all guess, but no one truly knows... at least not yet!

The Object of Our Affection

OK, OK, enough introduction! We've talked about history and spoke in generalities about object orientation, but I want more. The introduction was interesting, but I

really didn't learn anything meaty about object orientation. I'm ready to dig in and learn something. So, what is an object, anyway?

Well, let's start at the beginning. OO technology is fundamentally based upon, what else, the object. Turn your attention, for a moment, to a subject with which you should be familiar: entity-relationship diagramming. The data base modeler constructs an ER diagram that maps the entities in the real world together with the inter-relationships of those entities with one another. What is an entity? It is a real-world "thing" of some sort or another about which your computer-based application needs to record information. Well, an object can be thought of as a complex entity.

But, before we go any further, let me define a term I will be using throughout this article: classic. The term classic refers to the way things are currently done in the procedural world of corporate data processing. For example, when I speak of a classic DBMS implementation, I mean a DBMS traditionally used by business today. This can mean CODASYL/network (IDMS), hierarchic (IMS), inverted list (ADABAS) and even relational (DB2, Oracle). A classic application therefore would be a procedural application (possibly using a classic DBMS) based upon the principles of structured programming.

So, back to objects. In a classic DBMS, a logical entity is transformed into a physical representation of that entity solely in terms of its data characteristics. Be it an IMS segment or a DB2 table, data elements are stored within the physical representation of the entity. These data elements describe the current state of that entity. By contrast, an ODBMS would define an entity in terms of both its state and its behavior. In other words, an object encapsulates both the data (state) and the valid procedures that can be performed upon the object's data (behavior). See Figure 1.

Realize, however, that the true definition of an object is much more complex than this. I have purposely simplified this discussion to introduce the notion of encapsulation. The definition as stated does, however, introduce the basic difference between OO and classic methodologies. Think for a moment in classic terms: Many procedures are required to operate upon independent data structures. Each procedure must retrieve the data, operate upon it in some way, and possibly replace the data. In the OO paradigm, messages are passed to objects invoking the encapsulated methods. Because each object contains its own operations, or methods, most of the procedural code is eliminated and reusability is increased.

Remember, though, we are still talking basics. We have entirely avoided features that further enrich objects such as classes and inheritance. But, even though we have kept the discussion at a very basic level thus far, we have learned a very valuable component of the OO paradigm. Just remember that "an object encapsulates both data and process" and you will have learned your first OO lesson!

Class Is in Session

I think I get the basic idea. Let's see, instead of designing the data and procedure portions of our applications separately, with OO techniques we design them together. But we don't stop there. We actually implement them that way—together. But you said that we had just scratched the surface of objects. What did you mean? Is there a lot more to know?

I should hope so! Object-oriented technology is neither as complicated as people make it sound nor as simple as just understanding the concept of encapsulation. We must delve deeper into the core of what is implied by the term "object."

Let's start with a simple question: How are objects defined? An object is much more than just a data store encapsulated with its related procedures. It is a means of grouping real-world "things." In the real world, things are not simple. Data administrators discovered this long ago. That is why data models are deemed to be so important. It is simply not viable to implement an application system without first systematically and methodically examining the characteristics of the data that the system must support. Object-oriented technology takes this a step forward by making it important to not only model both data and process before proceeding, but modeling them both together.

Importantly, object-oriented technology enables objects to be defined based on the characteristics of other objects. Quite often one real-world thing is very similar to another in both composition and function. However, some (or many) of their qualities may differ in significant and meaningful ways. Are these objects the same? Should they be physically imple-

Object-Oriented Terminology

Behavior - refers to the manner in which objects change over time.

Class - the mechanism used for defining the data elements and methods for a particular type of object. All of the objects within any given class will have the same composition and behavior, but the state of each can be different (based upon the data in their variables at any given time).

Class Hierarchy - a tree structure delineating the relationship among classes. A class hierarchy can be composed of any number of levels and any number of classes, but only one top-most class.

Encapsulation - the technique of combining data together in a single, common area. This creates an environment in which all of the operations for a given set of data are organized and maintained in one place, thereby reducing confusion, eliminating misuse and simplifying maintenance.

Inheritance - the technique whereby variables and methods from higher level classes within a class hierarchy are available to be used by lower level classes.

Message - a request to an object to perform a method.

Method - a process encapsulated within an object.

Object - a representation of a real-world thing encapsulating the data and all of its procedures (processes) within itself.

State - the "make up" of an object at a given point in time.

Figure 1: Encapsulation

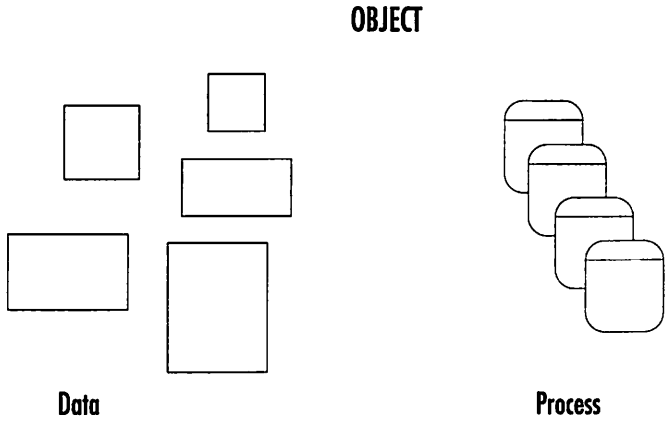
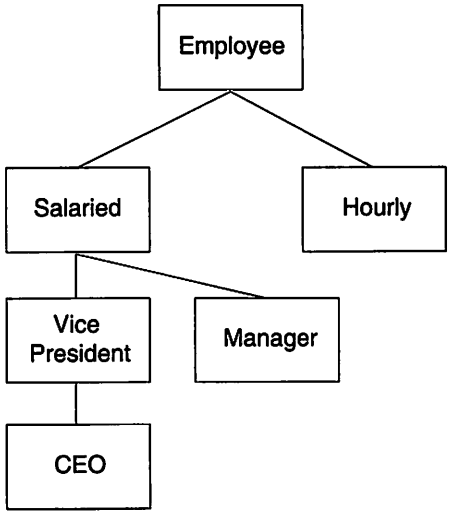


Figure 2: Example Employee Hierarchy



mented and stored separately or not? Classic implementation techniques have not generally agreed upon methodology for creating similar, yet different, entities. Some may be implemented physically within the same data store; others separate. In some instances, a combination of separation and combination may be used. This results in a confusing physical implementation. Object-oriented technology uses a single technique known as inheritance to implement objects.

Let's learn by example. Consider the task of trying to model the employees of an organization. In any organization, all employees exhibit many common characteristics (job duties, a salary of some sort, employee identification, etc.). However, based upon the type of job that they perform, employees can also take on quite different characteristics. Figure 2 is an example of different types of employees.

A data model would define the "what" of these types of employees. It would probably employ some type of hierarchy to depict these relationships. An object-oriented model embodies both data and process. Further, the implementation of the object-oriented model implements both data and process together.

It is quite obvious that each of these categories can be treated differently. However, they are also all employees with certain shared attributes and functions. How should this be implemented? The object-oriented paradigm uses classes to implement this hierarchical delineation of objects quite elegantly.

All objects are based upon a class. Each of the employee types in Figure 2 would be implemented as a class. The class would define the attributes and methods for objects assigned to that class. So, in retrospect, when we stated that objects encapsulate data and process, we should have said that classes provide this encapsulation. Figure 2, therefore, can also be said to represent a class hierarchy of employees.

What is gained by having classes? To explain this, we must define another OO concept introduced earlier: inheritance. Subordinate classes in a hierarchy inherit the variables and methods of superior classes. Simply stated, every class has all of the variables and methods of higher level classes available to it. So, those things which are common to employees are stored at the highest level. Those things which are specific to a certain type of employee are stored at lower levels within the class hierarchy. This enables each class to be similar, yet distinct.

You can see that a significant amount of modeling activity is necessary to derive benefit from this implementation. You must understand which procedures and data elements apply to which class before you develop your OO applications. With a good model and object-oriented techniques, you can derive the benefits of reusability, thereby reducing the length of the application development life cycle.

Synopsis

Hopefully, we have built a foundation upon which further object-oriented concepts and techniques can be discussed. Remember, there is much more to object orientation than the basics we have covered in this article. For example, OO technology also incorporates complex ideas such as abstract data types, polymorphism, overloading, multiple inheritance and persistence. The sidebar on Object-Oriented Terminology (page 15) will help you review and learn the basics. Future issues of *Data Management Review* will review object-oriented technology.

Craig S. Mullins is a member of the Technical Advisory Group at PLATINUM technology, inc. He has more than seven years of experience in all facets of data base systems development, including developing and teaching DB2 classes, systems analysis and design, data base and system administration, and data analysis. You can contact the author via Prodigy [WHNX44A] or Compuserve [70410,237].

■ *Was this article of value to you? If so, please let us know by circling Reader Service No. 31.*