CHAPTER 9

OBJECT-ORIENTED & MULTIMEDIA DBMS

9.1 INTRODUCTION

Recent advances in technology have made object-oriented & multimedia DBMSs possible. What follows is a set of relevant papers as they relate to object-oriented technology & engineering data management.
ELEMENTS OF
OBJECT-ORIENTED
PROGRAMMING

BY GEOFFREY A. PASCOE

What features must a language have
to be considered an object-oriented language?

THERE ARE AS many different views
of what object-oriented programming
is as there are computer scientists and
programmers. Because the term was
first used to describe the Smalltalk
programming environment developed
at Xerox PARC (see references 1–5),
this article will present object-oriented
programming concepts, terminology,
and characteristics from that perspec-
tive. I will stress the differences be-
tween an object-oriented program-
ing style and the more conventional
procedure-oriented style. I hope to
show that a language must have four
elements to support object-oriented
programming: information hiding,
data abstraction, dynamic binding,
and inheritance. (Some languages that
have one or two of these elements
have been improperly called object-
oriented.) I will follow this with a
discussion of the advantages and dis-
advantages of object-oriented pro-
gramming languages.

DATA PROTOCOLS VS.
OBJECT MESSAGES
Most programming languages sup-
port the "data-procedure" paradigm.
Active procedures act on the passive
data that is passed to them. A typical
example might be a square root func-
tion sqrt(x), that takes a number and
returns its square root.

In a strongly typed language such
as Pascal, it would be typical to have
a different version of sqrt(x) for each
data type of x, usually returning a
floating-point result. A late-binding
language such as LISP detects x's data
type at run time and performs the ap-
propriate operations for that data
type. Such generic operations are
generally primitives restricted to a
small class of data types such as
numbers, or they are functions de-
finied in terms of such primitives.

Object-oriented languages employ
a data or object-centered approach to
programming. Instead of passing data
to procedures, you ask objects (data)-
to perform operations on themselves.
To emphasize this difference I will use
an object-oriented expression syntax
of my own invention. The object name
is followed by :operation, followed by
any further arguments, and ter-
minalized by a period (in some ways
the syntax is similar to that of Small-
talk-80, though simpler). For example,
an expression to take the square root
of x will look like this:

x :sqrt.

The implication is that x is asked to
perform the :sqrt operation on itself.
You say that x is the receiver of the
message :sqrt.

A more complicated example would
be the dot-product operation. The
dot-product of two one-dimensional
vectors, x and y, is computed, produc-
ing a scalar result:

x :dot y.

Here, x is told to perform a dot-
product operation with itself and
the message :dot y and y assign the result to the
variable z in the following manner

z ← (x :dot y) :sqrt.

(continued)

Geoffrey A. Pascoe (1027 Scituate Harbor,
Pasadena, MD 21122) is a researcher in the
Department of Defense Computing Systems
Research Division and moderator of the
Smalltalk conference on BIX.
using parentheses to indicate the order of evaluation, although the parentheses are really not needed here, assuming left-to-right evaluation.

**SOME TERMINOLOGY**

The basic terminology of object-oriented programming is illustrated in the text box “Data Structures” on page 142. The object that I have been referring to (i.e., x, in the sqrt example) is an instance of a class. The class provides all the information necessary to construct and use objects of a particular kind, its instances. (For this reason, a class is sometimes called a factory.) Each instance has one class: a class may have multiple instances.

The class also provides storage for methods. Methods are simply procedures that are invoked by sending selectors to a class’s instances (also called sending messages). The methods reside in the class to save storage, since all instances of a class have an identical set of methods. Methods may allocate temporary variables for use during the execution of the method. These temporary variables are much like local variables in Pascal procedures in that their value is lost when you leave the method.

Each instance has storage allocated for maintaining its individual state. The state is referenced by instance variables. Instance variables may be primitive data types such as integers, other objects, or both, depending on the language. Each object has its own set of instance variables. Both temporary and instance variables may be freely referenced within the scope of an object’s method, but unlike temporary variables the value of instance variables is not lost when you leave the object’s method.

Computation is performed by sending messages to objects, which invokes a method in the object’s class. Typically, a method sends messages to other objects, which invokes other methods, etc., until you reach the point where a primitive method is invoked. Here ends the chain of message-sends. Each message-send eventually returns a result to the sender (e.g., x^sqrt returning the square root of x). The end result of all these message-sends is usually the changing of the state of one or more objects. Sometimes, however, a message is sent simply to invoke some primitive having a side effect external to the world of objects, for example, accessing an external file system or controlling hardware.

To fully support object-oriented programming a language must exhibit four characteristics: information hiding, data abstraction, dynamic binding, and inheritance. I will examine two languages, Ada and Modula-2, that have been mistakenly called object-oriented in order to illustrate why all four characteristics are necessary and why conventional procedure-oriented languages cannot adequately support object-oriented programming.

**INFORMATION HIDING**

Information hiding is important for ensuring reliability and modifiability of software systems by reducing interdependencies between software components. The state of a software module is contained in private variables, visible only within the scope of the module. Only a localized set of procedures directly manipulates the data. In addition, since the internal state variables of a module are not directly accessed from without, a carefully designed module interface may permit the internal data structures and procedures to be changed without affecting the implementation of other software modules.

Most modern languages, even FORTRAN, to some degree, support information hiding. ISO (standard) Pascal is one notable exception, since it provides no way to declare static variables within the scope of a procedure.

**DATA ABSTRACTION**

Data abstraction could be considered a way of using information hiding. A programmer defines an abstract data type consisting of an internal representation plus a set of procedures used to access and manipulate the data. Modula-2 provides excellent data abstraction mechanisms. Listing 1 is a fragment of Modula-2 module that implements a stack as an abstract data type (called an opaque type in Modula-2 terminology. See reference 6). Variables of type Stack may be
declared and manipulated in other program units.

Modula-2’s data abstraction mechanism provides a certain degree of protection since no direct access to the internal state of a stack is provided. The stack is manipulated through the module’s processing and query procedures. But there are a couple of problems with the Modula-2 solution. One is that the procedures used by a module must have either unique or qualified names. For example, if a module uses (imports) two different abstract data types, Stack and Queue, and variables of these types must be initialized, then the initialization procedures defined for these types must have different names, such as InitializeStack and InitializeQueue, or their usage must be qualified, as in Stack.Initialize and Queue.Initialize. This makes the resulting program less versatile. But more important than the need for unique identifiers is the drawback that Modula-2 abstract data types can operate on only one type of data. Stack, therefore, can store only integers.

Ada partially solves both these problems through two language features: operator overloading and generic program units (see reference 7). Operator overloading permits a program to use multiple operators with the same name. The distinction between operators can be determined at compile time by examining the types and number of parameters. just as + can be used to add either integer or real numbers in most modern languages. Generic program units permit the definition of a module to be used with different data types. The generic program unit is a procedural template that can be parameterized with actual types during compilation of programs using its capabilities.

A problem still exists if you wish to use the stack to store heterogeneous elements. Neither compile-time solution, operator overloading or generic program units, is sufficient. The solution is dynamic binding.

**Dynamic Binding**

Dynamic binding is needed to make full use of this code for stacking other types of data. Consider the addition of a procedure. Print, to the StackHandler module that prints the contents of a stack. If you use the stack for storing integers, floating-point numbers, character strings, etc., a traditional procedure-oriented approach dictates that you include a case statement to check at run time that the correct printing procedure for an element’s type is used. Trying to print an integer with a procedure designed to print character strings is potentially disastrous. The resulting problem is that every time you add (continued)
new type of data to the software system, you must modify all such case statements and recompile—a time-consuming and error-prone procedure. Ideally, additions should require only additions, not modifications.

The object-oriented approach pushes the responsibility for printing elements onto the objects themselves. Each object is sent the exact same message selector, Print, so that it will print itself in the proper way. This is known as polymorphism, since the same message can elicit a different response depending on the receiver. Operator overloading in Ada does not exhibit this form of dynamic polymorphism since the address of the procedure invoked is fixed at compile time.

This model of object-oriented programming can be improved. As presented thus far, the addition of a new type of object requires writing entirely new procedures for common operations such as Print. What's worse is that there will be a great deal of mularity between different Print methods, requiring continual rewrites of methods that differ slightly or not at all. This burden is likely to be so great that programmers would avoid the creation of new object types, significantly reducing the practical usefulness of object-oriented programming systems. Inheritance is a mechanism that largely relieves programmers of this burden.

**INHERITANCE**

Inheritance enables programmers to create classes and, therefore, objects that are specializations of other objects. For example, you might create an object, Trumpet, that is a specialization of a BrassInstrument, which is a specialization of a WindInstrument, which is a specialization of a MusicalInstrument, etc. A Trumpet inherits behavior that is appropriate for BrassInstruments, WindInstruments, and MusicalInstruments.

Creating a specialization of an existing class is called subclassing. The new class is a subclass of the existing class, and the existing class is the superclass of the new class. The subclass inherits instance variables, class variables, and methods from its superclass. The subclass may add instance variables, class variables, and methods that are appropriate to more specialized objects. In addition, a subclass may override or provide additional behavior to methods of a superclass. Methods are overridden when you provide a new method for an old method's selector.

The mechanism to add new behavior to an existing method tends to be language-dependent. In those languages most closely modeled after Smalltalk, this is accomplished by embedding a message-send to the supermethod in the new definition of a method. The mechanism is the same (see the text box "Pseudovariables" on page 144). For example, suppose you have an initialization method, initialize, defined in a superclass. If a subclass adds some instance variables, x and y, that must also be initialized to zero, then both initialization behaviors will be exhibited by the following method:

```plaintext
initialize  
  super initialize.  
  x ← 0.  
  y ← 0.  
```

The initialization in the superclass is performed, followed by the added initialization behavior. Depending on the placement of the message-send to super, the new behavior may precede, follow, or surround the existing behavior.

Inheritance enables programmers to create new classes of objects by specifying the differences between a new class and an existing class instead of starting from scratch each time. A large amount of code can be reused this way.

**ADVANTAGES**

Object-oriented languages have many advantages over more traditional procedure-oriented languages. Information hiding and data abstraction increase reliability and help decouple procedural and representational specification from implementation. Dynamic binding increases flexibility by permitting the addition of new classes of objects (data types) without having to modify existing code. Inheritance coupled with dynamic binding permits code to be reused. This

(continued)
Figure A: Data structures illustrating concepts related to object-oriented programming. Arrows indicate pointers.

A pointer to its class and a field indicating the length of the instance. The pointer to the class is needed so that the class, and therefore an object's methods, may be easily located when the instance is sent a message. Storing length information in the instance aids the storage manager and is useful when dealing with indexable instances.
as the attendant advantage of reducing overall code bulk and increasing programmer productivity, since you have to write less original code. Inheritance enhances code "factoring" (see references 1 and 2). Code factoring means that code to perform a particular task is found in only one place, and this eases the task of software maintenance.

**DISADVANTAGES**

Object-oriented languages have a few characteristics that are considered disadvantages by some. The one most often debated is the run-time cost of the dynamic binding mechanism. A message-send takes more time than a straight function call. Some studies have shown that with a well-implemented messenger this overhead is approximately 1.75 times a standard function call (see references 8 and 9). Actual differences in execution speed between traditional languages and their object-oriented counterparts, however, do not prove to be very significant. This is most likely due to the fact that the overhead applies only to message-sends and that message-sends accomplish more than a function call. Often, some of the work done automatically by a message-send must be done by the programmer anyway using code surrounding function calls or even multiple function calls. In fact, a case can be made that in large applications the ability to standardize and fine-tune the functionality supplied with the message-sends can make the application run faster than a traditional counterpart. The primary reason is that messaging obviates much of the variability in function setup code that results from different programming styles and skill levels. Messaging also eliminates the complex code often needed when traditional programs have to simulate dynamic binding.

Another disadvantage often cited is that implementation of object-oriented languages is more complex than comparable procedure-oriented languages, since the semantic gap between these languages and typical hardware machines is greater. Therefore, more software simulation is required. Fortunately, you pay the cost of implementation only once for a given machine.

Another potential problem is that a programmer must learn an often extensive class library before becoming proficient in an object-oriented language. As a result, object-oriented languages are more dependent on good documentation and development tools such as Smalltalk-80 browsers (see references 2 and 4).

**CONCLUSION**

There are other important concepts in object-oriented programming that I haven't covered because they either do not fit well into the Smalltalk model or are not central to object-oriented programming. Two that require at least abbreviated mention are multiple inheritance and automatic storage management.

**Multiple inheritance** allows a class to have more than one superclass. The potential for code sharing is greater but the possibility of conflicts between multiple superclasses increases the complexity of such systems. Certain flavors of LISP get a great deal of power by using multiple inheritance.

**Automatic storage management**, though not necessary, is so useful that it almost qualifies as a fifth major element. Automatic storage management techniques such as reference counting and garbage collection permit programmers to ignore details concerning the release of an object's storage. Application code becomes cleaner, and the overall system becomes more reliable.

Object-oriented programming provides major advantages in the production and maintenance of software. Shorter development times, a high degree of code sharing (good factoring), and malleability. These advantages make object-oriented programming an important technology for building complex software systems now and in the future.

**REFERENCES**

1. Ingalls, Daniel H. The Smalltalk-76 Programming System Design and Implementation. Conference Record of the Fifth Annual Symposium on Principles of Programming Languages.

**PSEUDOVARIABLES**

Syntactically, pseudovariables are treated the same as normal variables. Their semantics, however, are different since they cannot be assigned a new value during any particular invocation of a method.

Two important pseudovariables in Smalltalk are self and super. Both self and super refer to the object that received the message currently being processed. For example, if there happen to be 22 different rectangle objects currently existing and one of them is sent the center message, then the system will set both the pseudovariables self and super to refer to that rectangle. The difference between these two pseudovariables lies in the way that the message lookup is performed.

When self is sent a message, the message lookup algorithm is identical to the way a lookup is performed when the message is sent externally, starting in the object's direct class. When super is sent a message, the lookup is performed starting in the superclass of the class in which the method is currently executing is found. Note that this is not necessarily the superclass of the object's class. This pseudovariable mechanism gives objects a controlled way of accessing superclass methods.