AIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910

301/587-8202

Centimeter

Inches

MANUFACTURED TO AIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

Reviews of Computing Technology:

Object-Oriented Technology (U)

by D. C. Skeen

WSRC-IM-90-83-9

March 1, 1993

Authorized Derivative Classifier

G. F. Hayhoe

Westinghouse Savannah River Company
P. O. Box 616
Aiken, SC 29802

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Table of Contents

Preface	1
1 Introduction	3
2 General Concepts	5
Why Objects?	5
Encapsulation	6
Classes and Instances	7
Inheritance	9
Message Passing	11
Overloading, Polymorphism, and Supering	12
Delegation	17
Protocols	20
3 Benefits of Object-Oriented Programming	23
Code Reuse	23
Code Maintenance	24
Code Quality	25
Case Studies	26
4 Unresolved Issues	29
Persistent Objects	29
Remote Objects	30

Documentation	30
Scalability	31
Tool Support	32
5 Industry Directions	37
The Object Management Group	37
Object-Oriented Extensions to the Relational Model	38
Object-Oriented Operating Systems	38
Third Party Support For Objects	39
Industry Use of OO Technology	40
6 Impact of Objects	43
Organizational Changes	43
The Need For Managed Data	44
The Developer Skill Set	45
End Users	46
7 Conclusions	47
Appendix A: The Developer Scenario	49
Appendix B: Glossary of Terms	53

Preface

At any point in time, the computer industry has a comparatively small set of buzz words which serve as a time-dependent barometer of "what's hot and what's not." Ask any vendor representative, and you will see that as soon as a new buzz word (for example, *buzz word x*) becomes active, it is only a matter of time before virtually every commercial product is buzz word x-compliant. Some examples of industry buzz words are "client-server," "repository," and "relational."

Those responsible for an organization's orderly evolution to emerging technologies are themselves constantly undergoing an evolution within a "scale of disbelief" with regard to industry buzz words. The first stage in the evolution is one in which the technology watcher believes everything his or her sales representative says. This stage is called "patsy." Fortunately, people in this category tend to have little or no power or influence. The second stage in the evolution is one in which the technology watcher believes that a large portion of what his or her sales representative says is true. This stage is called "optimistic." The third stage is one in which the technology watcher believes that a small portion of what his or her sales representative says is true. This stage is called "jaded." The final, most mature stage in the evolution is one in which the technology watcher thinks that none of what the sales representative says is true. This final state is called "manager."

One addition to the recent set of industry buzz words is "object-oriented." True to form, many commercial products are suddenly, or soon will be, object-oriented. There is a significant danger that those who need to keep abreast of important, emerging technologies will lump this new buzz word in with all of the others and treat it with whatever level of disrespect is dictated by their own evolution on the scale of disbelief. True, the term *object-oriented* has been, and will continue to be, oversold. True, much of the benefit of object-oriented technology is based upon as yet unfulfilled promises. True, there are many major unresolved issues with object-oriented programming. True, much of what is dreamed by those who dream object-oriented dreams will not come to pass. Nevertheless, it is important that organizations become aware of this technology, for it represents a fundamental shift in software development, management and use. The implications of this technology are far-reaching and potentially devastating for the unprepared organization.

This paper exposes the reader to the fundamental concepts embodied in object-oriented technology and offers some discussion of how this technology is likely to impact the development and user communities. Although some of this paper is fairly technical in nature, the reader is encouraged to make the effort to read carefully and fully understand its contents. For good or ill, whether you want them or not, objects are coming.

1 Introduction

A useful metaphor in introducing object-oriented (OO) concepts is the idea of a computer hardware manufacturer assembling products from an existing stock of electronic parts. In this analogy, think of the parts as pieces of computer software (called objects) and of the finished products as computer applications. Like its counterpart, the object is capable of performing its specific function in a wide variety of different applications. For example, just as a power supply could be used in many different pieces of equipment, a data access object can handle the data needs of many different applications.

The advantages to assembling hardware (and software) using a set of pre-built parts are obvious. The design process is greatly simplified in this scenario, since the designer needs only to carry the design down to the chip level, rather than to the transistor level. As a result, the designer is free to develop a more reliable and feature rich product. Also, since the component parts are reused in several different products, the parts can be made more robust and subjected to more rigorous testing than would be economically feasible for a part used in only one piece of equipment. Additionally, maintenance on the resulting systems is simplified because of the part-level consistency from one type of equipment to another. Finally, if a sufficient demand exists for a particular kind of part and standards exist for the requirements of that part, it may be possible to purchase the part from one of several different vendors. Thus, free market pressure will ensure the assembler that parts of high quality can be obtained at a reasonable cost, without committing precious internal development and manufacturing resources.

The remainder of this document introduces the techniques used to develop objects, the benefits of the technology, outstanding issues that remain with the technology, industry direction for the technology, and the impact that object-oriented technology is likely to have on the organization. While going through this material, the reader will find it useful to remember the parts analogy and to keep in mind that the overall purpose of object-oriented technology is to create software parts and to construct applications using those parts.

This page left blank intentionally.

2 General Concepts

Like any branch of computer technology, object-oriented (OO) design and programming extracts a price from its initiates in terms of the overhead of learning a new vocabulary. Unlike many other technologies, however, critical OO terms are not, for the most part, simply new words for old concepts. Instead, these terms represent concepts with no clear counterpart in more classical application development terminology. This fact is partly responsible for many misconceptions about object-oriented technology. This section will introduce the reader to most of the important terms needed to understand object-oriented technology, as well as provide some framework for understanding how these concepts relate to one another.

Why Objects?

It is clearly inappropriate to develop a computer program that would require the end user to enter "a little bit of COBOL code" for the program to perform its intended function. So strong is the inhibition against such a program that virtually all computer languages are compiled. That is, they consist of a pre-defined set of instructions that are cast in stone before the program is ever run. In other words, most computer languages do not even allow the *concept* of such a program. This state of affairs recognizes that it is unreasonable to expect everyone who needs the services provided by a computer to be an expert in the design, development, and maintenance of the systems that they use. Management expects end users to be proficient at their jobs, and end users expect the computer to be a simple-to-use adjunct to those jobs. In fact, careers have been built on the creation of better ways to shield end users from the internals of the programs that they use. The pervasive term *user-friendly* is the embodiment of this idea.

Unfortunately, the same effort has not been expended to provide developers with a simple-to-use interface. Instead, developers are expected to handle all the facets of programming needed to create a given application. This is not a problem as long as the programs to be developed are fairly simple and do not require proficiency in a number of different computer subfields. However, today's user has a very high level of expectations from the machine. Graphical

user interfaces (GUIs) are expected as a matter of course. Cross network connection to a collection of heterogeneous databases is demanded. Sophisticated multimedia applications loom on the horizon. Applications are expected to be delivered on a number of different platforms. Each of these items requires a specialized knowledge, and it is already to the point that it is unreasonable to expect a developer to be an expert in each of the subfields necessary to generate such applications.

Object-oriented development is the practice of creating programs that are developer friendly as well as user friendly. Object-oriented technology extends the idea of a simple-to-use interface, not just to the application as a whole, but to each of the parts that compose the application. In other words, just as programmers create applications that only require users to dictate what needs to be done (not how to do it), object-oriented technology is a set of tools, techniques, and conventions that allows application developers to specify what needs to be done (not how to do it).

Encapsulation

The observation has been made that if you ask 10 different object-oriented developers for a definition of objects, you will get 20 different answers.¹ This statement, while whimsical, is fundamentally correct. There exists no concise, universally agreed upon definition for what constitutes an object. One of the simplest definitions states that objects are programming units that combine data with procedures that operate on the data. While this definition omits some important characteristics of objects, it stresses the most important idea that, in the object world, data and all actions performed on that data are combined into a single, indivisible unit. This idea is called *encapsulation*.

Encapsulation has several important benefits. Among these is the high degree of abstraction that it affords. In a traditional programming scenario, coding is highly dependent upon the form of the data. A change in that form dictates changes in all applications that manipulate or use the data. For example, if some of the information about an employee that was previously contained in a single table is split into two or more tables, all of the code that accesses this data (from potentially many different applications) must be modified to accommodate the change. It is also likely that many different applications perform the same routine operations over the same data. This leads to a cornucopia of implementations of the same routine, all of which are implemented to varying levels of correctness and all of which must be maintained.

1 Bill Caffery speaking at a Gartner Group Conference, Symposium '91.

By contrast, reusing an encapsulated object in many different applications eliminates this duplication. It provides a single point of maintenance, as well as allowing applications that use these functions to continue to run without modification. In effect, the object defines an interface that applications use to perform whatever operations are necessary on that object. This shields the applications (and the application developer) from the underlying detail of how to accomplish a particular function. In this environment, application developers have the freedom to build more complex (and at the same time, more robust) applications, since they can focus on defining what to do, rather than how to do it.

Classes and Instances

Central to the idea of OO technology is the concept of a *class*. A class is a blueprint for the creation of objects. Objects created using that blueprint are *instance objects* or *instances* of that class. For example, a reasonable class for an organization to create might be a class called EMPLOYEE. This class would be the template for the definition of all data that pertains to an employee and all of the operations that are performed on that data. Each employee in the organization would then be represented by a particular instance of that class. The situation becomes somewhat confused since a class is, in and of itself, an object (said to be a *class object*). From the previous discussion, an object contains both data and operations performed on that data (the operations are called *methods* in object-oriented terminology). The class, then, contains two types of data definitions. One set pertains to the class itself as an object (the class data), and the other set pertains to the instances created from that class (the instance data). Similarly, the class contains two sets of method definitions. One set pertains to the operations performed by the class (the class methods), and one set pertains to the operations performed by instances of that class (the instance methods). A class object has access only to its class data and can perform only its class methods. An instance of that class has access only to its instance data and can perform only the instance methods.

As an example, imagine a blueprint for a house as a metaphor for a class and a house constructed from that blueprint as a metaphor for an instance. Some of the instance data associated with each house might be the kind of carpet that is used, the color of the house trim, or the question of whether to construct the fireplace of stone or brick. It is easy to imagine several houses, all of which are constructed from the same blueprint, each of which has different values for each specified instance data element. What is common among all instances of these houses is that they all contain the same data types but may have different values for those data types. There are also several activities that might be associated with the finished house, such as opening the front door, starting a fire in the fireplace, or putting furniture in one of the bedrooms. None of these methods would make sense to apply to the blueprint itself. They are all instance methods. Having the blueprint attempt to per-

form one of these operations would be nonsense. Some information, however, is pertinent to the blueprint, but not to the house. For example, the name of the architect may be inscribed on the blueprint, but the most thorough search of one of the houses will not turn up the architect's name etched in the baseboards.

Figure 1 describes a class called *HouseClass* that contains one class variable (*architectNameVariable*) and one class method (*architectNameMethod*). The *HouseClass* object, then, knows how to do only one thing (execute the instructions associated with the *architectNameMethod*). This class definition also contains two instance variables (*trimColorVariable* and *carpetTypeVariable*), as well as three instance methods (*openDoorMethod*, *buildFireMethod*, and *furnishRoomMethod*). Any of the three instances of houses constructed from these blueprints (John's House, Mary's House or Mike's House) can execute any the instance methods associated with the class (*openDoorMethod*, *buildFireMethod*, or *furnishRoomMethod*), and in doing so, can access its own unique instance data elements defined for instances of that class (*trimColorVariable* or *carpetTypeVariable*). None of the instances, however, can access any class methods or class data, and the class cannot access any instance methods or instance data.

Figure 1: An Example of a Class with Instances

In theory, this sounds confusing. In practice, it is not. Typically, the only operation performed by a class is to create an instance of that class. All other activity occurs in the instances of that class.² The fact that the primary purpose

of a class object is to create instances can be seen in the fact that class objects are sometimes called factory objects and class methods are called factory methods.

Inheritance

Classes do not exist in a vacuum. Instead, they are part of a class hierarchy, such as the one shown in Figure 2. In this figure, all classes are a subclass of the class OBJECT. OBJECT, then, is the *root class*. Since OBJECT is a class, it contains class methods and data, as well as definitions of instance methods and data. OBJECT has two child classes: EMPLOYEE and VEHICLES. Each of these classes, in turn, contains class methods and data, as well as definitions of instance methods and data. Since both the employee class and the vehicle class are subclasses of OBJECT, they automatically include all the class methods and data defined in OBJECT. That is to say, the class EMPLOYEE can perform both class methods defined in EMPLOYEE and class methods defined in OBJECT. In addition, the class EMPLOYEE contains any class data elements defined in OBJECT, as well as the class data defined in employee. Similarly, any instance of EMPLOYEE can perform, not only instance methods defined in EMPLOYEE, but any instance methods defined in OBJECT and contains any instance data elements defined in either OBJECT or EMPLOYEE. The ability to absorb the characteristics of a parent is called *inheritance*.

Figure 2: An Example of a Class Hierarchy

- 2 Obviously, the creation of an instance must be a class method. Otherwise, there would be a serious chicken vs. egg problem for creating the first instance.

Inheritance is important because it provides for a tremendous amount of code reuse. If a similar concept applies to both exempt and non-exempt employees (as in Figure 2), then that concept should be defined within the **EMPLOYEE** class. This cuts in half the number of times the code must be written, as well as providing a single point in the hierarchy in which to perform maintenance of that concept. Inheritance is a two edged sword. While pushing a concept up, the hierarchy increases its applicability, and that fact may require generalization to the point that the code necessary to implement the concept becomes quite complex.

In Figure 2, every class (except for the root class) has exactly one direct parent class. This is known as *single inheritance*. Some object-oriented environments allow a given class to have more than one parent class. This is called *multiple inheritance*, as shown in Figure 3.

Figure 3: An Example of Multiple Inheritance

In the real world, an object often belongs to more than one class of things. Figure 3, we see an example. Suppose that a government contractor maintains a fleet of cars and trucks purchased with government funds. These cars and trucks are certainly vehicles. As such, there is a need to work with the characteristics of these objects that relate to their role as a vehicle. Examples might be information about the license number, mileage, number of doors, etc. Since these vehicles belong to the government, there is also a need to deal with them as government property. For example, the price paid, information pertaining to audits, date acquired and so forth. Multiple inheritance allows for the creation of objects that can act in multiple roles. One might argue that making government property a subclass of vehicle would accomplish the same thing. This is true, but then all of the work that goes into defining the things that pertain to an object's "government propertyness" will apply only to vehicles. Treating other types of objects (computers for example) as government property would require a duplication of that effort. By the same argu-

ment, it would be improper to make vehicle a subclass of government property (the government does not own all vehicles). One of the differentiating factors among object-oriented environments is whether multiple inheritance is allowed.

Message Passing

Object-oriented systems establish flow of control by sending messages to objects. The syntax of message passing varies between OO environments, but all messages have at least three components:

- a reference to the intended recipient of the message
- the method that the sender wants the recipient of the message to execute
- optionally, any arguments that may be required for the specified method (optional because some methods require no parameters)

Message passing is analogous to function or procedure calls in a conventional program, with the important difference that some OO systems can resolve references at run-time, rather than at compile time.³ This is called *late binding*.

When considered in the context of an event-driven application, late binding allows applications to dynamically alter their behavior in response to a series of events. By contrast, the conventional developer must "hard-code," in program logic, the situations under which it is appropriate to execute one piece of code over another. The interested reader with access to a piece of source code for a conventional program is invited to scan the code to determine how much of the code performs some desired function and how much simply acts as a "traffic cop." The percentage of conventional code that performs this activity is often surprisingly large, especially when considering that this is the most error prone part of most programs. This sort of programming logic is almost non-existent in OO applications. The OO application, instead, relies upon response to (often user-generated) events to drive the flow of control of an application and can dynamically determine what messages should be sent to what objects in response to the current state of the application.

3 Some conventional programming languages do allow the programmer to execute any of the functions which have been compiled into the program and to delay the selection of which functions to execute until run-time (that is, passing function references as parameters). However, even in languages that allow the possibility, it is sufficiently unusual that the feature is almost never exercised, and this is more restrictive than the generalized idea of late binding.

While messages are indeed analogous to function calls in more conventional programs, it is perhaps more accurate to compare them with Remote Procedure Calls (RPCs). RPCs are a mechanism whereby a piece of code on one machine can cause code on some other machine to execute some action.

Messages are a mechanism whereby one object can cause another object to execute some action. The fact that these objects may be physically located on different machines is usually of little or no consequence to either object.⁴ This makes OO techniques ideal for implementing client-server (CS) systems since the cross-network nature of CS systems can be made nearly transparent to the developer.

An important thing to note about messages is that a crucial part of the message is the reference to the intended recipient. It is incumbent upon the sending object to somehow determine the recipient of a given message. Literature that gives a high-level summary of objects usually overlooks this point. The reader is often left with the impression that undirected messages are "sent out into the ether" and, through some magical process, arrive at the doorstep of the intended receiving object. In fact, messages are sent only to specific targets.

Overloading, Polymorphism, and Supering

The ability of classes to inherit characteristics from a parent class is obviously a useful characteristic, but it can be quite difficult to write methods that are generic enough to handle every case. It seems that, no matter what kind of system is being developed, there are always exceptions. An object-oriented concept called *overloading* provides a way of handling this situation.

Figure 4 shows two classes: EMPLOYEE and a subclass of EMPLOYEE called EXEMPT EMPLOYEE. Normally, instances of the class EXEMPT EMPLOYEE would inherit their method for calculating pay from the instance method described in the class EMPLOYEE. However, since the class EXEMPT EMPLOYEE defines a method by the same name, instances of the class EXEMPT EMPLOYEE will use the EXEMPT EMPLOYEE definition of

-
- 4 Some difficulties do occur if a parameter is passed by reference from one object to another. In this case, the address referenced by the sending object is in a different address space than that used by the receiving object. This problem can be solved by allocating space on the target machine, copying the contents of the source reference to that newly allocated space and passing the reference to that space as the parameter to the target object. It is possible to do this in a manner that is transparent to the sending object and nearly transparent to the receiving object.

the method instead. In other words, when competing method definitions exist in an object hierarchy, overloading dictates execution of the method defined in the class that is most specific (nearest parent) to the object.

Figure 4: An Example of a Class Hierarchy with Overloading on the Method "calculate pay"

Suppose, for example, that there exists a particular instance of an employee called "Pat" and a particular instance of an exempt employee called "Chris." Both of these instances can calculate their pay, but each one does it in the way that is appropriate for instances of their respective classes. The beauty of this mechanism is that it not only abstracts applications that need to invoke this method from the need to know how the method accomplishes its task, but from the need to know which class the instance belongs to as well. That is, from the perspective of the application, instances of either class are simply told to "calculate pay"; it is the responsibility of the instance in question to accomplish that task in the appropriate way. The ability of different objects to respond to the same message in different ways is called polymorphism.

Overloading is not the only way that polymorphic behavior can be obtained. Figure 5 shows two object classes: REACTORS and BATTERIES. Each of these classes has a method called CHARGE, but each of these methods performs a very different activity when the CHARGE method is invoked. The object SOMEOBJECT can send a CHARGE message either to the instance ASPECIFICREACTOR or to the instance ASPECIFICBATTERY. The action that will take place is as much a function of the recipient of the message as it is the message itself.

Figure 5: An Example of Polymorphism without Overloading

Overloading provides a mechanism for replacing a method defined at one level in the hierarchy with one defined at a lower point in the hierarchy. The concept of *supering* provides for an augmentation to an existing method,⁵ rather than the replacement of an existing method. Supering occurs when a polymorphic method, as part of its operation, invokes a method by the same name that it inherited from its parent class. As an example, Figure 6 shows an alternative method of handling the calculate pay problem.

In this case, NONEXEMPT EMPLOYEE has been defined as a subclass of EMPLOYEE, rather than the opposite as in the previous example. Chris is still an instance of an EMPLOYEE (that is, an exempt employee) and Pat is still a nonexempt employee. Since Chris is an instance of EMPLOYEE, Chris will use the instance method defined for the class EMPLOYEE (and therefore will receive no overtime pay). The instance Pat, being an instance of the class NONEXEMPT EMPLOYEE, will use the instance method defined in the class NONEXEMPT EMPLOYEE. However, the first action of the NONEX-

⁵ The concept of supering does not exist in all object-oriented environments. In those environments where the concept does exist, the term *supering* is not always used.

EMPT EMPLOYEE method is to super to the method defined in the EMPLOYEE class. The effect of this is that the first thing that the NONEX-EMPT EMPLOYEE method does is to calculate the regular pay (that is, without overtime) for the nonexempt employee. The NONEXEMPT EMPLOYEE method then augments the pay by adding any overtime pay due the Pat instance.

Figure 6: An Example of Supering

One of the most common uses of supering is to perform initialization of an object.⁶ Consider the example shown in Figure 7. Each class in this example hierarchy contains a single instance variable that must be initialized. Each class also defines an instance method called INITIALIZE. The instance BOB is of CLASSC, so it initializes using the method defined in CLASSC. The first action in that method is to super to the INITIALIZE method in CLASSB. This first step of the instance INITIALIZE method for CLASSB is to super to the instance INITIALIZE method in CLASSA. The INITIALIZE method from CLASSA is performed (setting the value of the instance variable VARIABLEA to 0) and the INITIALIZE method from CLASSB continues to execute and initializes the instance variable VARIABLEB. When this is complete, the INITIALIZE method for CLASSC continues to execute and initializes the instance variable VARIABLEC.

Figure 7: An Example of Initialization Using Supering

⁶ It is also commonly used for reading data, writing data, and destroying objects.

At first, it may seem cumbersome to initialize in this way. After all, it is quite simple to define the initialization method for CLASSC to perform the initialization of all three variables and eliminate the extra code (and overhead) of supering. There are several reasons why this would be undesirable. In the first place, pushing the initialization of the variables of CLASSA down to some lower level would be inappropriate since instances of CLASSA would then be unable to initialize (they would have to be of CLASSC to get access to the initialization code). Therefore, the code to initialize the variables in CLASSA must exist as a method within CLASSA. Furthermore, even if the developer could somehow be certain that no instances of CLASSA will ever be created, there is always the possibility that some new subclass of CLASSA could be created. If so, code to initialize the CLASSA variables will have to be replicated for the new subclass. Code replication is one of the primary things that object-oriented programs seek to avoid. Finally, pushing the initialization down to CLASSC creates a dependency between CLASSC and CLASSA. In order to add a new variable to CLASSA or to modify the initialization of a variable in CLASSA, the modification will have to occur in CLASSC. The foremost principle in object-oriented programming is that anything that pertains to a given class be defined within that class and nowhere else.

Delegation

Objects respond to messages from other objects, but it is not necessarily the receiving object that performs the requested work. It is possible for an object to receive a message and then have some other object perform the actual work. This is called *delegation*.⁷

Delegation has several uses. First of all, delegation can be used as a version of multiple inheritance within systems that support only single inheritance. That is, an object can simulate the characteristics of an additional class by creating an object of the other class and allowing the newly created object to handle any messages that pertain to that class.

In Figure 8, some external object requests the property number for some vehicle by sending a message to that vehicle object (through the message [SomeVehicle propNumber]). The vehicle object receiving the message handles it by telling its delegate (SomeGov'tProperty) to deal with the message.⁸

7 Delegation is not universally implemented in every object-oriented system but is fairly simple to implement, even in systems that do not support the concept directly.

8 This example assumes that the object-oriented environment being used does not support the concept of delegation directly, since the developer of the vehicle object had to specifically write code (albeit simple code) to

Figure 8: An Example of Using Delegation
for Multiple Inheritance

In an OO system that supports late-binding, the delegate for a particular object can be a variable and may change many times during the run of an application. Delegation, then, allows an object to adapt its behavior to a particular situation.

In Figure 9, there exists an object (SOMEOBJECT) which wants a particular image displayed on the screen. The problem is that there is no single standard for representing images. Instead, there are multiple, competing standards for image representation. The actions needed to display the image depend upon which format the image is in. One possibility for handling this situation is to create multiple types of image classes, require applications that want to work with images to be cognizant of what format each image is in, and work with an image object of that type. A more developer-friendly solution, however, is to create an object (ANIMAGEOBJECT) which handles any type of image. This object has several methods (METHODA, METHODB, and METHODC) which are the same, no matter what kind of image is being manipulated. Methods that are format dependent are developed in separate objects (ATIFFIMAGEOBJECT and ANEPSIMAGEOBJECT), one for each supported image type. When ANIMAGEOBJECT needs to manipulate a

handle a message intended for the delegate. In a system which supports the idea of delegation, the message is routed to the delegate automatically.

TIFF⁹ image, it sets its delegate to be ATIFFIMAGEOBJECT. If, on the other hand, an EPS¹⁰ image is being handled, it sets its delegate to be ANEP-SIMAGEOBJECT. Generic image methods are handled by ANIMAGEOB-JECT, while format dependent methods are handled by ANIMAGEOBJECT's delegate. From the viewpoint of SOMEOBJECT, it appears as though ANIMAGEOBJECT can handle any type of supported image.

Figure 9: Using Delegation for Adaptive Behavior

The level of abstraction afforded by delegation is so strong that future additions to the set of supported image types can be made and used by the external object without any modification to the external object. In other words, an image application that supports image formats that do not even exist at the time the application is written can easily be developed.¹¹

9 Tagged Image Format File

10 Encapsulated PostScript

11 Depending upon the environment being used, the application may or may not have to be relinked to utilize new formats.

Protocols

Objects interact with one another by passing messages. For one object to send a message to another object, the sending object must be assured that the receiving object is able to respond to that message in the desired way. One way of doing this is through *protocols*. A protocol is a set of messages to which a particular object must respond. An object which responds to that set of messages is said to adhere to that protocol. It is important to note that a protocol is a *definition* of the messages to which an object must respond, not an *implementation* of those messages. Protocols are a mechanism for the developer of OBJECTA to specify the methods that the developer of OBJECTB must provide in order for OBJECTB to work with OBJECTA. Figure 10 shows the definition of a simple protocol called CONTAINER. The CONTAINER protocol requires two methods, SETVALUE:aValue and GETVALUE. Figure 11 shows a subclass of USERINTERFACE called TEXTBOX which implements both of the methods defined in the CONTAINER protocol. The TEXTBOX class, then, adheres to the CONTAINER protocol.

Protocol Container

```
    setValue:aValue;  
        /*Makes the current value of the receiving object aValue*/  
  
    getValue;  
        /*Causes the receiving object to return its current value*/  
  
end
```

Figure 10: A Simple Protocol

Figure 11: An Example of an Object Class
Which Responds to a Protocol

If an object (OBJECTA) which performs some useful service (for example, reading and writing object values to and from a database) exists, it may be desirable that this useful service be applied to objects of the TEXTBOX class. Furthermore, assume that OBJECTA is able to perform its useful function on any object that adheres to the CONTAINER protocol. Then objects of the class TEXTBOX can be successfully manipulated by OBJECTA. In other words, a protocol is a definition of the messages to which an object must respond in order to be manipulated by some other object. The protocol is defined by the object doing the manipulation but implemented by the object being manipulated. It is not strictly necessary that the SETVALUE:aValue and GETVALUE methods be defined within the TEXTBOX class for TEXTBOX to adhere to the CONTAINER protocol. It is perfectly permissible for TEXTBOX to inherit one or both of those methods from some parent class.

If the implementation of the SETVALUE:aValue method (Figure 11) is examined, it is the first line of code (`myValue = aValue;`) that does the work required by the CONTAINER protocol (that is, an internal variable, MYVALUE, is made equal to whatever value is passed in the parameter AVALUE). The second line of code (`[self display:myValue]`) is not necessary to support the protocol but is used to support its basic function as a USERINTERFACE object (that is, to display its current value to the user).

Some object systems provide formal support for protocols, while others do not. In those systems where formal support exists, one object can be queried by another object to determine whether it adheres to a particular protocol. While formal protocol support is desirable, protocols can be used in OO systems which do not directly support protocols. That is, object class developers can be required to provide two methods for every object class developed.¹² The first of these methods would allow the object class to register the protocol(s) to which the class adheres. The second method would be invoked by other objects to determine whether or not the receiving object adheres to a given protocol. Even in systems which do not provide support for protocols (either as part of the language or because of programmer discipline), they are still important. A protocol defines the interface that one object expects another object to provide.

12 These two methods would then constitute the "protocol protocol." If the reader understands this statement, the reader probably understands protocols.

3 Benefits of Object-Oriented Programming

Code Reuse

The benefits of this technology are usually easier to demonstrate than they are to explain. Any comparison of object-oriented techniques with conventional programming methods will inevitably degenerate into "I could have done that with conventional methods." The point is correct. Ultimately, code is code. Anything can be done in assembler if you want it badly enough. The point is also moot. Conventional applications are simply not developed in a way that the advantages of code reuse are inherently designed in. The fact is that applications are developed as custom solutions to specific problems, with little or no thought given to how the effort put forth to develop that application can be reused in the future. Current code reuse usually takes the form of vendor-supplied libraries, with no ability for the developer to extend the capabilities of those libraries for incorporation into future applications. In other words, like it or not, today's application developer must constantly "reinvent the wheel."

Another form of code reuse that exists in conventional development is of the "cut and paste" variety. That is, a developer finds a previous application that performs some desirable function and copies that portion of the code into a new application. The reasons that code cannot be simply transferred from one application to another are many, but two reasons stand out. The first is that applications are state- or context-dependent. A given section of code cannot be removed from the total application environment without maintaining the state of the program as determined by other parts of the code. Secondly, code is dependent on the data structures over which it operates. A given piece of code will only operate in a different application if that new application provides an identical set of data structures. Writing "modular" code does not improve this situation. A module in a functionally decomposed application relies heavily on communication (through a well-defined interface) with other modules to perform certain operations which are necessary for that module to operate. That module, then, cannot be moved to another application without

taking its supporting modules with it. A domino effect which makes it difficult to move one module without moving all of the modules ensues.¹³

This limitation is intrinsic to monolithic applications. Object-oriented applications, by contrast, are developed as a loosely coupled collection of stand-alone entities (objects) which contain all the data necessary to represent a particular concept, as well as the operations performed on that data in support of that concept. Given an appropriate set of objects, the application developer creates an application by selecting an appropriate subset of those objects and invoking some portion of their preprogrammed behavior by passing appropriate messages to those objects. In other words, the application developer tells the objects what to do, not how to do it. This allows the developer to be much more productive, both in terms of rapid application development and in terms of application development that involves technologies outside the developer's normal area of expertise.

The key point is that code reuse is not enabled by OO because of some magic associated with the technology, but because the technology lends itself to a different development approach. Instead of focusing on a never-ending stream of specific applications, developers focus on the common objects that need to be manipulated to run the business. These objects are reusable in many different applications.

Code Maintenance

Virtually any discussion of software engineering will point out that programmers devote a significant portion of their effort to maintenance. The estimates of percentage of effort vary, but most agree that maintenance constitutes over half of the life-cycle cost of an application. When developers focus on solving an organization's software needs by concentrating on providing solutions on a problem by problem basis, there will necessarily be some overlap as different applications perform similar actions. This is not only inefficient from a development standpoint, it is counterproductive from a maintenance standpoint. Should some external change occur, each application must be modified to accommodate that change. Object classes, on the other hand, provide a single point from which all applications draw upon the data and methods defined for that object. Therefore, when maintenance needs to be performed on an object class, the worst case is that applications which use that class will need to be recompiled.¹⁴

13 This is not a condemnation of modular programming. Modular programming offers a host of benefits, but reusability is not one of them.

14 This is, admittedly, a rather idyllic view of objects. In some cases, a modification to an object class will result in new functionality that an

A problem that is at least as disturbing as the extra effort needed to maintain multiple versions of the same action is that the software maintenance task can be so daunting that the organization is unable to incorporate a superior technology. For example, it may be impossible to use a new database management system or a better network service because of the potential impact on existing locally developed software. The single point of maintenance afforded by OO allows an organization to examine the effort needed to improve the computing infrastructure and to act accordingly without fear of a ripple effect causing paralysis. OO technology, when seen in this light, is not simply a preferred method for developing and maintaining software, but an improved vehicle for change management as well.

While OO techniques reduce the amount of necessary maintenance, they do not eliminate the need for maintenance entirely. There will always be end user requests for additional features or other changes. The conventional developer begins this process by obtaining a thorough understanding of the existing code, either by extensive experience with the application or by climbing a rather tedious learning curve. The OO developer, on the other hand, can subclass an existing object class to create a new object class that has all the characteristics of its parent class. The developer can then use overloading and polymorphism to replace existing methods with new ones, use supering to augment existing methods, and create new methods to perform new functions. Augmenting existing methods and creating new methods can often be done with little or no knowledge as to how a particular feature of the existing class is implemented. Note that this modification can occur even without the source code for the existing classes. This allows the OO programmer to even customize the object classes provided by vendors in a comparatively simple way.

Code Quality

There are other advantages to the paradigm besides increasing programmer productivity. While OO technology does indeed increase the *quantity* that a developer can generate, it may be more important that it also increases the *quality* of what the developer generates. A conventional developer must be a jack-of-all-trades. For the portion of an application that is a user interface, the developer puts on a user interface hat. Similarly, for the portion of the code that deals in database access, the developer puts on a database hat. If an application deals with the network, then the developer puts on a network hat. Today's application developer must have a large hat rack. It stands to

existing application may want to take advantage of. In this case, the application may have to be modified, although it should continue to work as usual without any problems.

reason that as developers must develop proficiency in more and more areas, the depth of knowledge in each area decreases. In the OO paradigm, however, individuals who are expert in the particular subject area develop each class of objects. As an example, database experts develop database access objects, GUI experts develop GUI objects and so on. The job of the application developer is to interconnect these prebuilt objects in a desired way. The application developer, therefore, leverages the expertise of each specialist.

This approach has the side effect of promoting adherence to software standards. By developing a set of common objects upon a base of accepted standards, applications that use these objects automatically adhere to those standards. In other words, an organization can enforce standards not only with the stick of mandate, but with the much more compelling carrot of ease of development. Standardization in this way also allows end users to enjoy a common "look and feel" among locally developed applications and reduces the amount of training required to use those applications.

Case Studies

If these OO claims sound a little too good to be true, consider a Gartner Group case study of a resource allocation and tracking system developed for American Airlines by Sabre Computer Services using OO technology.¹⁵ Using a function point comparison between developers of this system and the baseline expected of Sabre developers for conventional development, these techniques made the inexperienced OO developer 2.18 times as productive as a conventional counterpart. After building a first application, the developer became 2.63 times as productive as the conventional counterpart. After the initial investment in building the class library, the OO developer becomes 3.30 times as effective as a conventional counterpart, and in the area of user-interface is 3.78 times as prolific as the conventional counterpart. Lest the reader become concerned that this breakneck development pace comes at the expense of code quality, consider that this system replaced a conventional system containing about 100,000 lines of code. The testing phase of the OO system uncovered only two bugs. If the reader is concerned that this speed comes at the cost of system flexibility, consider that changes to system requirements late in the development process required that the system be modified from a workstation-centric application to a client-server application, a change that was effected in less than a month, or that extending the server to handle multiple, simultaneous clients took less than a day.

15 *Gartner Group Research Note, Software Engineering Strategies, "American Airlines Implements Mainstream Object Systems,"* May 15, 1992.

These results are not exceptional. There are many examples of highly successful OO systems.¹⁶ One example is the Maintenance Management System (MMS) developed for General Motors by EDS. The MMS schedules over 10,000 preventative and corrective maintenance jobs per month, tracks maintenance history for all of the machines, and manages the inventory of spare parts. The OO system was developed over 3.5 months to replace an existing PL/I system which was developed over a period of 19 months. This is an improvement ratio of better than 5:1. The original program required 13.6 megabytes of main memory to operate, while the OO version required only 1.1 megabytes, an improvement of better than 12:1. In terms of simplified maintenance, the original code consisted of 265,000 lines of code, compared to the 22,000 lines of code for the OO version, an improvement of more than 12:1. Most impressive of all is the reduction in developer resources required to complete the two projects. The original system required 152 staff months to complete, while the OO application required 10.4 staff months to complete, resulting in an improvement of over 14:1.

Perhaps the best known use of OO technology is the Customer Management System (CMS) developed by Brooklyn Union Gas in conjunction with Anderson Consulting. This system was developed to replace an existing system developed 13 years earlier and was used by 80% of the company's employees in support of over one million customers. Despite substantial enhancements to the original program, the resulting system is 40% smaller than the original. The system is accessed by 850 employees and generates 40,000 bills, 80,000 credit activities and 250 reports every day. The system contains nearly one million lines of code and manages a 100 gigabyte database. The entire support staff for this system consists of 12 people.

One of the most curious examples involves a system that is still in the design/prototyping phase. This is Texas Instruments' Semiconductor Computer Integrated Manufacturing (CIM) System. The unusual thing about this project is how the estimate of the number of lines of code required for the finished system keeps dropping as the project continues. The initial estimate for the project was 500,000 lines of code. The current estimate is less than 200,000 lines of code. This is clearly counter to conventional systems development, where the estimate almost inevitably increases as the project continues.

16 These examples (and many others) can be found in *Object-Oriented Information Systems: Planning and Implementation* by David Taylor (John Wiley & Sons Inc., 1992).

This page left blank intentionally.

4 Unresolved Issues

Persistent Objects

In the conventional programming world, applications are separate from the data that they manipulate. The applications themselves are temporal. They exist only while they are running. The data, on the other hand, is less ephemeral. Knowledge workers are used to being able to retrieve data from a database or scan a file, whether the applications that generate and manipulate that data are running or not. This presents something of a problem in the object-oriented world, since the data is indivisibly bound with the code that operates on that data. Like an application, the object only exists when it is resident in memory. An object is *persistent* if it is—or appears to be—always available for execution.

There are several ways to make an object persistent, but this is typically the domain of object-oriented database management systems (OODBMS). These systems direct queries against stored representations of objects and instantiate those objects that meet the qualifications of the query. These objects can then be sent messages as usual to perform some desired behavior. That is, an OODBMS stores both the data associated with an object and the methods that it is capable of responding to, and instantiates the object on demand. Unfortunately, few if any OODBMS are of “industrial strength.” A great deal of work needs to be done before they are ready for commercial use. Most industry analysts agree that this functionality will, for the most part, be incorporated into more conventional relational database management systems over the next few years.¹⁷

The reader should not construe the persistence problem to mean that OO systems are incapable of storing data in a conventional DBMS for later retrieval. In fact, it is quite simple to make an object store its instance variable values

¹⁷ There is a discussion of object-oriented extensions to the relational data model in a later section.

in a DBMS as a matter of course and to make this action transparent to any developer using the object. Several tools exist to simplify this process.

Remote Objects

In many cases, it is desirable for each application running on each workstation in a network to have its own collection of instances that belong to and reside on that workstation. Examples certainly include the user-interface objects used by the application and may include others as well. Some objects, on the other hand, need to be shared. For example, in an application that accesses or manipulates "employee" objects, it is not necessarily desirable that all 20,000+ employee instances be instantiated on each workstation running the application. Among other reasons, coordinating updates to the instance variables for those objects would be a nightmarish task. Instead, it would be desirable to run those objects remotely on an "object server." The normal database issues of concurrent update then come into play. In large part, this remains an unsolved problem, but standards are emerging in this area. The main effort in this area is the Object Request Broker standard recently put into place by the Object Management Group (OMG).¹⁸ This standard is too recent to have generated commercially available products that adhere to it, but most of the significant object vendors are members of OMG and have pledged to support the standard.

Documentation

Typically, developers document two things about a program: the algorithm(s) and the implementation of the algorithm(s). Documentation of this sort is appropriate for describing the implementation of a particular class, just as it is in a conventional system. However, unlike conventional programs, this documentation needs to be associated with the object class, not an application that happens to use an object of that class (anymore than a conventional program should document the algorithm and implementation used by a "cube root" routine that the program happens to access from some library).

There are two reasons for this. First, developers should be able to use (not develop, use) objects that perform some function *without* an intimate knowledge of how to implement that function. Forcing the developer to document that function in every application that uses it is counter to this idea. Second, if

18 The Object Management Group is discussed in the Industry Trends section of this paper.

class documentation exists in every application that uses an object of that class, then the documentation for each application must be updated every time the class is modified. Otherwise, the documentation will quickly become outdated.

Object *class definitions* require approximately the same kind of documentation that exists in conventional programs today (albeit, with a strong emphasis on the interface), but OO *applications* must capture a different kind of information. The OO application developer constructs applications by connecting a series of objects in some way in order to fulfill some business need. What the documentation must capture is the business problem and the interconnection used to solve that problem. Unfortunately, this interconnection is often difficult to describe on paper. While a few diagrams will go a long way toward describing the relationships between objects, an exhaustive list of the interconnections is virtually impossible. OO environments typically mitigate this problem by providing some sort of browser to allow the developer to determine what connections exist between objects. It is ironic that the developer can establish connections in a few seconds that may take several minutes or even hours to describe.

The point of all of this is that OO technology introduces the need for an additional kind of documentation. This documented information, while suitable for conventional documentation methods at a high-level, ultimately requires the use of tools running against the source code to fully extract the details. This is counter to conventional documentation practice.

Scalability

There are many successful examples of object-oriented systems that are in production today and delivering benefit to the companies that use them. Nevertheless, there are virtually no examples of an organization that has built its entire core computing infrastructure around objects. Even NeXT computer, generally regarded as being in the forefront of OO technology, does not use OO development in their core business systems (they instead provide object interfaces to more conventional business system engines).

Code reuse is the primary benefit of objects, and code reuse only realizes its full promise when two or more applications can share a common object hierarchy. It is one thing to develop an object class to the point where it is satisfactory for use by a single application. It is something else again to develop that object class so that it is suitable for many different applications. As an example, consider an object class for employees called "Employee." The requirements for the Employee class (from the perspective of a work management application) are quite different from the needs of, say, some application in the medical department. OO advocates would quickly point out that requirements for both systems must be supported no matter what technology is used and that OO techniques eliminate the redundancy in that which is common

to both systems as well as providing a single point of maintenance. The fact remains, however, that no one has ever created such an object hierarchy on a large scale. Whether this can be successfully done has been neither proved nor disproved, and scalability remains an outstanding issue.

Tool Support

A number of options are available to a developer wishing to develop object-oriented applications in terms of available tools. Unfortunately, all of the tools have serious limitations in their current states. The closest thing to an industry standard for workstation development is C++. While some versions of this product do have a development environment associated with them, that environment is rather limited in its power and scope. Moreover, the object hierarchy included with that environment is scanty at best. Reliance on this product would require the local development of many basic object classes.

Another alternative is found in the "pure" object environments, such as Smalltalk, Actor, and Eiffel. These environments do provide a fairly strong development environment. On the downside, only Smalltalk works on a variety of platforms, and Smalltalk suffers from a lack of standardization across those platforms. Even more of a drawback is that their inherently "pure" object approach will complicate the process of integrating applications written for these environments with conventional systems. Finally, while vehement proponents exist for each of these environments, the industry trend is away from these languages and toward the C-based ones.¹⁹

In the interests of completeness, another possibility is the Lisp-based object-oriented development environments, notably the Common Lisp Object System (CLOS).²⁰ While the CLOS development environment is certainly a powerful one, movement toward this environment as a standard would, to say the least, represent a considerable departure from anything that the company has done to date. Besides the stigma of being Lisp based, this environment does suffer some portability problems, and the maintainability of Lisp-based systems has long been an issue for debate.

19 There are those who would disagree with this assertion. There is little doubt that C++ sales are proceeding at a much faster pace than sales of Smalltalk. However, the dominance of C++ is largely in the areas of system programming, real-time transactions, and the scientific and engineering areas. The presence of either language in office application development is miniscule, but the edge is held by Smalltalk.

20 Pronounced "See-Loss."

Finally, there are the high-end object development tool sets such as NeXTstep, and to a lesser extent, the suite of software development tools for the Sun Workstation.²¹ Both of these offer a powerful development environment and, in the case of NeXTstep, a fairly robust set of objects as a development base. Both of these environments, however, function only within a proprietary environment (NeXTstep for the NeXT workstation and NeXTstep/486 for the IBM/486 in the case of NeXTstep, and Sun workstations in the case of the Sun tool suite).

In short, today, there exists no perfect object-oriented development environment. Fortunately, relief can be expected from two different directions. All the vendors that supply the products described above are members of the OMG. It is anticipated that the OMG will succeed in developing standards that will solve many of the cross-platform issues that have been described. Additionally, the emergence of a set of industry-wide object standards will lead to the suitable augmentation of any provided object hierarchies that would otherwise be too weak to be considered as a base for object development. A second and perhaps more likely standard is the joint Apple/IBM Taligent venture. Both Apple and IBM are members of the OMG, and, ideally, the Taligent effort will result in an object standard that is compliant with the OMG.

Object-oriented programming is an emerging technology and the current state of the tools reflects this. An organization should not, however, sit by and wait for the tools to reach "industrial strength" before making a serious effort to understand and use this technology. This paper recommends that development groups begin working on pilot applications and small scale production applications using the best available tools to train developers in the paradigm and to begin to understand the nature of the needed object hierarchy. Later, when the toolsets do become more robust, development can proceed into full scale production with vendor support for an OMG standard, if possible, and by reimplementation under a de facto standard, if necessary.

The table below compares various OO environments by certain features. One of the distinguishing characteristics of an OO environment is whether it is interpreted or compiled. In an interpreted system, source code is transformed into executable code as it is being executed. In a compiled system, all of the translation is done prior to the execution of the code. Both types of systems have their advantages. Compiled code is typically much more efficient, since the translation overhead is not required at run-time. Interpreted systems, on the other hand, generally offer greater flexibility and ease of development to the developer.

21 Strictly speaking, the Sun development tools are not object-oriented per se but are adaptable to object-oriented use.

Characteristics of Common Object-Oriented Tools					
	Compiled or Interpreted	Typing	Binding	Inherit	Environment or Language
Smalltalk	Either	Weak	Dynamic	Single	Environment
C++	Compiled	Strong	Static	Multiple	Language
NextStep/ ObjectiveC	Compiled	Strong	Dynamic	Single	Environment
Eiffel	Interpreted	Strong	Dynamic	Single	Environment
Object Pascal	Compiled	Strong	Dynamic	Single	Language
CLOS	Either	Weak	Dynamic	Multiple	Environment

Another basic characteristic of an OO system is whether it supports strong or weak typing. In a strong typing system, the type of a variable must be determined prior to its use. Weak typing systems allow variables to be of any desired type. Again, there are advantages to either approach. Strong typing systems have the advantage of being able to perform checks to make sure that parameters are of the proper type and generally allow for more optimized code. Weak typing systems, on the other hand, allow greater flexibility in the program. In fact, weak typing systems can be built to handle variable types that do not exist at the time the program is created.

OO environments are also differentiated by whether they support static or dynamic binding. Static binding requires that all message references be resolved at compile-time, while dynamic binding allows references to be resolved at run-time. Dynamic binding does induce a modest performance overhead, but the flexibility of dynamic binding far outweighs this cost.

Some OO systems support multiple inheritance (object classes can have more than one parent class), while others support only single inheritance. Again, there are merits to either approach. Allowing an object class to inherit from multiple parents clearly increases the potential for code reuse. Unfortunately, multiple inheritance creates conflicts between competing polymorphic methods that burden both the developer and the OO system.

Some OO tools are simply languages, while others are full-blown development environments. Again, there are trade-offs. The advantage to a language-only tool is that the developer can use whatever development environment is appropriate. On the other hand, an integrated development environment typically provides more powerful tools for dealing with a given language. Also, since all application developers share a common environment in this paradigm, the probability for exchanging objects is higher.

The following table gives the author's opinion of the relative ranking of various OO tools with regard to several important characteristics. A 1 indicates that a particular environment is "best" in a particular category, while a 5 indicates that it is the "worst." Ties are permissible.

Relative Ranking of Various OO Development Environments					
	Platform Variety	Toolset Strength	Class Library	Ease of Integration	Consistency with Industry Direction
Smalltalk	1	2	1	4	3
C++	2	5	5	1	1
NeXTStep/ ObjectiveC	5	1	2	1	2
Eiffel	3	3	4	5	4
CLOS	4	3	3	3	4

This page left blank intentionally.

5 Industry Directions

The Object Management Group

The Object Management Group (OMG) is a consortium of major computer manufacturers and object vendors, including IBM, DEC, Apple, NeXT, Borland International, Sun Microsystems, Software AG, and others.²² The OMG is somewhat unusual in that it is a standards group formed by the industry before proprietary standards could proliferate. In a sense, OMG was formed as a pre-emptive strike against a host of mutually incompatible standards throughout the industry. While many analysts doubt the long term viability of the OMG, it is currently a growing organization that has managed to attract most of the major players, and its release of the Object Request Broker (a standard for remote object requests) must be hailed as a significant accomplishment. The Object Request Broker is a framework for "services for naming objects, versioning objects, managing object interface definitions and implementations, providing atomic sequences of requests, controlling concurrent requests, providing security and notification of concurrent events."²³

The Object Request Broker is the first of many pieces of the OMG's Object Management Architecture (OMA). OMA is a high level architecture that addresses interoperability issues for OO systems. Other pieces include the Object Model and Object Services that provides "functions for realizing and maintaining objects [and] are critical to building inter-working, portable clients and object implementations."²⁴

22 At the time that this paper was written, OMG was composed of over 200 member companies.

23 "Object Services RFI Released," *First Class: The Object Management Group Newsletter*, February/March 1992.

24 "Object Services RFI Released," *First Class: The Object Management Group Newsletter*, February/March 1992.

Object-Oriented Extensions to the Relational Model

Most industry analysts do not believe that object-oriented database management systems are going to become the normal vehicle for storing and retrieving objects. Instead, most believe that vendors will make extensions to the relational database management systems (RDBMS) to allow them to handle objects. An indication of this trend can be seen in the fact that more traditional database vendors, such as Oracle and Software AG, are active participants in the OMG. There are several reasons that extensions to RDBMSs are the most likely vehicle for object storage. The most obvious reason is that the investment industry has made in conventional database technology is simply too significant to throw away. This is true of the training and application investment, as well as the cost of the DBMSs themselves. Less obvious, but no less important, is the fact that the RDBMS vendors themselves have made a tremendous investment in perfecting their products in the areas of backup, maintenance, and security. These are areas that the newer OODBMS vendors have not had enough time to develop to the same level as their RDBMS counterparts, even though these areas are critical for acceptance by DBMS professionals.

Object-Oriented Operating Systems

It is perhaps ironic that object-oriented operating systems have been so late to emerge, considering how object-like they are to start with (the process control block contains data pertaining to the process and the kernel performs operations on that data). At any rate, there has not been a great deal of successful effort in this area. The need for object-oriented operating systems is obvious. Object-oriented systems today are somewhat stymied because they are forced to operate in a non-object environment. That is, "object orientation" must stop at the boundary of operating system services. One of the problems that this creates is that much of the functionality necessary for object systems to work (for example, support for message passing) must be provided by the object systems themselves. For the most part, this restricts objects to niche, stand-alone systems, with very little contact with other types of systems. Another problem that this induces is that every OO environment develops its own way of providing necessary services. Consequently, OO environments are less interoperable than they could be.

Taligent promises to solve these problems. Taligent is a cooperative effort between IBM and Apple with the goal of producing an object-oriented operating system that runs on both platforms.²⁵ Initial release of the software to the original equipment manufacturer (OEM) is scheduled for 1993, and release to

the computer community at large is scheduled for 1995. Two major points about the system are already known. The first is that system services will be provided by a set of system object classes. The second is that the protocols to which the system objects adhere will be published and available to the computing public at large. As a result, Taligent offers an unprecedented ability for third party developers and industrial developers to tailor operating system services to meet specific needs. Moreover, since the basic protocol is well-defined, organizations should be able to "mix and match" various system service objects from various vendors with reasonable assurance that they will work together in harmony.

There are also nearer term OO operating systems. Among these is PenPoint, an object-oriented operating system for pen-based computers developed by GO Corporation (although the PenPoint kernel is not object-oriented). PenPoint provides the developer a rich set of over 250 object classes in categories such as Window/User-Interface, Application management, Text and Handwriting analysis, and File System manipulation. PenPoint supports message passing at the kernel level.

Third Party Support For Objects

It might be argued that the "acid test" for a new technology is whether or not it creates an industry of third party support for itself. In the case of object-oriented programming, the third party industry is still in its infancy but will continue to make significant growth. As examples, consider that numerous third party companies are developing class libraries in support of Borland's C++.

Perhaps the most significant example of third party object support is related to NeXTstep. NeXT Computer, Inc., has recently released the ObjectWare catalog of over 100 commercially available objects from over 50 third party vendors. These objects are fully compliant with the OMG's Object Model and range from general purpose user interface objects to highly specific objects for stock trading. Additionally, a nationwide effort called "Project Object" undertaken in the summer of 1992 promises a wide variety of public domain objects in the near future. The importance of third party object support cannot be overemphasized. No single end-use organization can possibly hope to develop and maintain all the objects necessary to support the organization. Third party developers offer a way for an organization to use high-quality ob-

-
- 25 It might be argued that a system which provides cross-platform compatibility alone is sufficient reason to move to objects for organizations with a heterogeneous workstation base.

jects in their applications without tying down most of their development organization in developing and maintaining those objects.

Industry Use of OO Technology

A recent informal industry survey of companies indicated the following statistics about the use of OO technology.²⁶ Roughly three-quarters of the companies surveyed are at least experimenting with object technology. Nearly one-third of the companies have object applications in use or under development. This is shown in Figure 12.

Figure 12: Industry Use of OO Technology by Percentage

Figures 13 and 14 refer to the current and planned acquisition of OO tools, respectively. It may be interesting to note that the decline between current and planned acquisitions for Smalltalk is much steeper than the decline between current and planned acquisitions for C++. This is an indication that C++ is

²⁶ Gartner Group Research Notes, *Applications Development and Management Strategies*, "ADM Clients Evolving from OO Explorers to OO Developers," September 8, 1992.

on its way to becoming the dominant language of choice for OO applications. It is also interesting to note that OO Analysis and Design Tools (which were considered as part of "Other" in Figure 15) is the largest category of planned acquisitions in Figure 16. This is an indication that OO users are moving out of the experimental phase and into the development phase for OO applications and are consequently more interested in addressing the complete life-cycle of OO applications.

Figure 13: Percentage of Current Acquisition of OO Tools
for Companies That Have Purchased Such Tools

Figure 14: Percentage of Planned Acquisition of OO Tools
for Companies That Plan Such Acquisitions

6 Impact of Objects

Organizational Changes

Object-oriented programming represents a fundamental change in application development. It is not surprising, therefore, that organizations responsible for program development may need to evolve as well. Paramount among these anticipated changes is the need for a new type of developer called an "object architect." The responsibility of an object architect is to develop and maintain locally developed objects and to ensure that new and modified objects interact with harmony rather than with discord. "Software reuse is a complex discipline which may follow several alternative schemes, but will achieve optimal results with a centralized group chartered with responsibility for class libraries."²⁷

Since object classes can be subclassed and modified in many different ways by different developers, object architects need to be aware of these changes to elevate generic modifications into the base classes so that the depth of the object class hierarchy remains at a manageable level. Object architects will typically have a broad understanding of the global object hierarchy, be extremely proficient in object-oriented analysis and design and have deep expertise in one or more computer technology domains (database, networking, graphics, etc.).

27 *Gartner Group Research Notes, Applications Development and Management Strategies*, "ADM5: Managing the Transition to OO Development, July 10, 1992.

The purpose of the object architecture is to empower the application developer with a set of pre-built, pre-tested components from which to build applications. While it is the intent of the object architecture to provide as much functionality as needed to create any given application, it is unlikely that such a complete hierarchy can ever be constructed. Therefore, it is incumbent upon the application developer to:

- use the existing hierarchy of objects to implement an application whenever possible
- be able to create application-specific objects within that hierarchy when required

It will also be necessary for application developers to realize when a required change to the existing hierarchy is application specific (that is, should be handled by an application-specific object) and when a change is caused by a deficiency in the hierarchy (that is, should lead to a modification of the existing hierarchy by the appropriate object architect). Training in object-oriented development and design, therefore, cannot be limited solely to the object architects, but must be provided to all developers.

A surprising side effect of code reuse is that some way must be found to pay for it. This is particularly true in a zero-based budget environment. The development of reusable objects is naturally more time-consuming than implementing the same thing in a way that is good enough for a particular application. A customer with a need that requires the development of a new object may be reluctant to pay for the extra development time needed to create the object in a generic manner. After all, the original customer does not benefit from the reuse of the object in a subsequent application.²⁸ In at least one case, an organization has handled this problem by considering the creation of an object as an investment on the part of the customer funding the original development of the object and paying a "royalty" on the use of that object in subsequent applications. While this may be somewhat extreme, management does need to formulate some policy to handle this situation.

The Need For Managed Data

It has been mentioned repeatedly throughout this document that an object is the encapsulation of data and the operations that are performed on that data. An object hierarchy, then, cannot be created until there is a clear vision of the data needs of the organization. Moreover, there must be agreement on a con-

28 One could argue that the customer receives the benefit of reusable objects from the fact that at least some portion of the application can be developed from the objects created for previous applications.

sistent view of that data. This is a necessary precursor to developing the object hierarchy. In fact, the analysis of corporate object needs will likely begin by looking at the company data model. There are many reasons why an organization needs a consistent, managed data administration function. Object-oriented programming is simply one more reason why this should be done.

The Developer Skill Set

The largest single problem in transitioning to object-oriented technology is that it requires a fundamental shift in the mind set of developers and analysts. They must learn to look at a problem in terms of the entities being manipulated in the context of the entire organization, rather than developing a sequential, step-by-step solution to a series of specific problems. This is not the kind of thing that a developer can learn by attending a week or two of training. Instead, the developer must climb a rather steep learning curve. It is ironic that novice programmers often find it easier to learn to develop object applications than developers with more experience. The more experienced developers have more to unlearn in the face a new paradigm. Fortunately, most experienced developers eventually catch up and pass their less experienced counterparts as they learn to apply their greater knowledge in a new way.

Because of the difficulty in making the transition to object-oriented thinking, it is imperative that the development groups begin experimenting with this technology today. This is the only way to obtain the necessary gurus to lead the organization toward the technology when it begins to mature in the latter part of this decade. Without a strong core of developers experienced in the technology, it is doubtful that objects can be successful, regardless of how the technology and tools develop.

In a similar vein, the organization may need to consider the importance of the C programming language. While languages that were designed to be object-oriented languages from the ground up exist (Actor, Eiffel, and Smalltalk, for example), these languages account for only a fraction of the object-oriented development that has been done to date, and the computer industry is clearly not heading in that direction. Similarly, while object-oriented programming can trace many of its roots to the Lisp-based object languages (CLOS and ROSS, for example), this is not where the industry is going. Instead, most OO development is being done in the C based languages, notably C++ and Objective-C. While C is quite popular for developing system software, process control applications, and scientific applications, its use in the business world today is limited at best. Despite the obvious power and flexibility of the language, opponents of C in the business arena have argued correctly that C is a "high-level assembly language," unsuitable to rapid business development, especially when compared to a 4th generation language (4GL). The introduction of objects into C, however, invalidates this argument. In fact, speed of development in current high-end C based object environ-

ments surpasses even the best of 4GLs, while retaining the inherent flexibility of the language. For this reason, the industry trend in objects is, and will continue to be, C-based.

This is not to say that the development of object-oriented applications will require all developers to be proficient in C. In fact, it is possible to develop object applications using absolutely no code in any programming language (C or otherwise). However, the majority of purchased objects will be implemented in C-based languages, and the simplest method of extending those objects and creating a hierarchy of locally developed objects that integrate with those objects will be to continue the use of the C-based languages. C programmers experienced in developing business applications are relatively scarce. Management may need to take a look at this problem and consider training needs and hiring practices accordingly.

End Users

At first glance, it may not seem that the use of OO technology would have much impact on the end user. After all, today's users do not typically care about the implementation details of the applications they use. There will be, however, three major impacts on the end-user community as a result of OO technology, all of them positive. The first of these is that end users will begin to see the same level of user-friendly interfaces to locally developed applications that they have come to expect from commercially developed workstation software. As a result, organizations can expect an increase in the productivity of the average information worker.

The second advantage is more difficult to explain, but at least as valuable. Given a common base object hierarchy, applications constructed upon that hierarchy tend to interoperate at a fundamental level. This is not as a result of intent of the application developers as much as it is that the applications rest upon the same foundation. For this reason, standardization on a corporate object hierarchy will lead to a much tighter level of integration between the products of third party vendors than has been the case.

The third impact is that there will be more commonality among the user interfaces to applications (both commercial and locally developed). This will increase user comfort with the use of the applications, as well as reduce the amount of required training. Along the same line, users can expect an increased level of application functionality provided as system services, as opposed to an application specific implementation. As an example, consider how spell checking is performed under NeXTstep. Spell checking is a system service in this environment rather than a function provided on an application-by-application basis. As a result, users of the environment need to learn only one spell checker and need to maintain only one user dictionary of unique words. Not only does this make the spell checker simple to use, the spell checking service is available to the user in any application, not just a word processor.

7 Conclusions

Object-oriented programming is an emerging technology that will not mature for several years. Nevertheless, preparation must be made today to take advantage of the opportunities afforded by the technology when it does mature over the next three to five years. The most significant preparations are to develop a corporate data management policy and to begin to develop the OO skills of a select group of developers to act as OO gurus when the toolsets improve. Objects will become an important part of the computing environment, irrespective of any action that organizations may or may not take, simply because the major software vendors are coming to the realization that objects offer the ability to develop software "better, faster, and cheaper" than the current technology. To compete in the market place, vendors will increasingly develop their software products using this technology. The opportunity to have these advantages for locally developed software is simply too good to pass up. The technology is not here yet, but it is most assuredly on its way, and the time to prepare for it is now!

This page left blank intentionally.

Appendix A:

The Developer Scenario

OO technology is fairly new, particularly in terms of its acceptance and use by most industrial developers. Therefore, it is probably appropriate to provide a scenario for how development might occur in an object-oriented environment. This scenario, while rather utopic, is based upon OO technology that is currently available.

Mary, an application developer, has been tasked to develop a vendor contact management application for the buyers in purchasing. As with any application, the first step is to understand the need that the client wants the application to address. So Mary sets up a meeting with the client to determine exactly what the needs are. During this meeting, the assemble-to-order nature of object-oriented "parts" provides a powerful mechanism for exploring the potential for the application with the user. Mary's interface to these objects is graphical in nature. That is, Mary is able to construct the application by manipulating a set of icons, rather than writing code. She rapidly constructs a rough prototype of the application while sitting side by side with the client. This gives both the client and the developer something concrete to focus on during the discussion. It also provides documentation of any decisions that are made. Mary and her client settle on two basic metaphors for the application. One metaphor is that of a data book which logs the time, date, recipient and subject of a particular call. The other metaphor is a rolodex containing the names, addresses and phone numbers of the vendors that the buyers work with.

Mary's next task is to assess the components that will be needed to complete the application in terms of components that already exist, components that can be created from existing components and new components. The user interface objects for the application already exist or can be easily subclassed from existing components. Many of the other objects, such as a database access object and a phone service object also exist, but it turns out that at least one new object will need to be created. This object concerns people who are points of contact for a given vendor. It seems that no application prior to Mary's has had a need to represent this information. Mary contacts one of the site object architects and discusses this problem. The object architect agrees that the

concept of a vendor contact is one that should be represented in the corporate object hierarchy, gets Mary's needs for the object from her and promises to deliver the object by the end of the week (the object architect will probably continue to develop the object after that time, but the basic functionality required for Mary's application is straightforward, and there is no need to hold her up while the object is fleshed out). Until the new object is ready, Mary will subclass the Person class to act as a proxy for the VendorContact class (that is, a vendor contact is a subclass of the class Person).

Mary proceeds with developing the application by polishing up the user-interface that she worked out with the client and connecting the objects to perform the functions that the application requires. This includes dialing a phone number selected from the rolodex, automatically logging the time, date, and contact for a particular call (the user must still supply the subject of the conversation), a lookup function to find a particular vendor or name within the rolodex, etc. This is easy for Mary to do since these actions are either already supported by the objects that she is using or require only a few lines of code to complete. Mary's biggest problem is the creation of a new class to manage the actual vendor contact information (the time, date, subject, etc.). While she could have left that to the object architect, the data is application-specific enough that she doubts it belongs in the corporate object hierarchy. Fortunately, this is a very simple object. It contains a data definition and two methods for each data element. One method sets the value of the data element, and the other method retrieves the value of the data element.²⁹ Mary must also create methods to store and retrieve this data from the database (after proper consultation with the Database Administrator, of course) and methods to obtain the current time and date. Mary creates this new class which she calls the VendorContactLog class. Within about a day, the application is basically complete, with the exception of integrating the VendorContact class. When the VendorContact class is delivered, Mary ties up a few loose ends and shows the finished product to the client.

A few days later, Mary receives a call from the client. The application is working fine, but an important requirement was never brought up in the requirements discussion. The application is based on the premise that vendor contacts are always initiated by a buyer, but many times it is the vendor that initiates the contact. This needs to be reflected by the application. Mary gives a sigh of relief since only minor modifications need to be made to accomplish this. Mary simply adds a new data field to the VendorContactLog class to reflect the originator of the contact and adds a couple of methods to manipulate the information. She promises to have the changes in before she goes to lunch.

29 The alert reader may have noticed that one easy way for Mary to do this is to utilize objects which adhere to the hypothetical CONTAINER protocol mentioned in the earlier discussion of protocols.

A week or so later, Mary receives another call from the client. The change is working fine, but it requires the buyer to launch the application in order to log the contact. That's no problem if the buyer is at his or her desk, but what if the buyer is out? It turns out that the purchasing department is using another OO application to act as a digital answering machine. As part of that application, the caller is allowed to punch in his or her telephone number (assuming that the caller is at a touch-tone phone). The client wants to know if there is any way to use that telephone number as an index into the rolodex and log the call.

Mary frowns because this one is a little tougher than the last one. Mary doesn't know anything about the telephone answering application, but the basic class that drives the application (Telephone) has become part of the corporate object hierarchy. Mary accesses the class definition (online) and is pleased to learn that the class developer adhered to the `ApplicationDelegate` protocol.³⁰ The purpose of the `ApplicationDelegate` protocol is to allow one object to be informed when an interesting event occurs in some other object. In this case, instances of the Telephone class can optionally be associated with an arbitrary object called the Delegate. If a Delegate exists, the Delegate is sent a message by the Telephone object whenever any of a set of interesting events occurs (or is about to occur). The message that Mary wants her application to react to is the "`IncomingMessageDidComplete`" message. That is, Mary wants her application to respond whenever the Telephone object records an incoming message. The only thing that Mary must do is create a new method in her application object that responds to an `IncomingMessageDidComplete` message and make her application object the Delegate of the Telephone object. The object id of the actual phone message is sent as a parameter to the Delegate, so Mary's method can ask the phone message (it's an object, so it responds to messages) what number was entered. Using that information, Mary's method then looks up that number in the rolodex and (if it finds a match) logs the contact (using the methods already defined by her application).

Mary makes the necessary changes to her class, but making her application object the delegate of the Telephone object is outside of the control of Mary's application. Mary contacts Jack, the telephone application developer, and asks that her application object be made the Delegate. This is a fairly simple process, but the fact that the two objects are in different applications does raise an interesting point. The problem is that an object must direct messages to a specific target object. If both objects are in the same application, this usually presents little problem. However, when the objects reside in different programs, some mechanism must be found to obtain the ad-

30 Mary doesn't know this, but the original developer of the class didn't think to adhere to the `ApplicationDelegate` protocol. Support for that protocol was added by an object architect when the class was elevated to corporate class hierarchy status.

dress of the desired object in the other application. One way to do this is to use a Name Service which allows one object to look up another object in much the same way that the telephone book is used. Mary and Jack agree on names for their objects and modify their code to register the object names with the Name Service. Moreover, upon initialization, Jack's object will attempt to locate Mary's application in the Name Service and make that his Delegate, if found. Upon initialization, Mary's application will attempt to locate Jack's application and will send a setDelegate message if Jack's application is found. In this way, Jack's application can set Mary's application as the delegate, regardless of the order in which they are launched.³¹

31 A more complex problem, which is beyond the scope of this paper, is the inherent problem of communication between asynchronous processes. While the event-driven nature of objects does simplify some of the difficulties when compared to conventional practices, the problems that arise from the unexpected failure of one of more of the processes remains.

Appendix B: Glossary of Terms

Abstract Superclass

A class which is not designed to directly create instances. Instead, the abstract superclass provides a common set of data and behaviors that are shared among its child classes (which may, in turn, create actual instances).

Behavior

Another term for method.

Category

A technique supported by some (not all) object-oriented environments that allows the developer to add additional functionality to an existing class (as opposed to subclassing that existing class). This is typically used to augment a vendor-supplied class with several child classes already defined. Rather than resubclass each of the child classes, the developer associates a category with the parent class and all of the child classes subsequently inherit the new methods. Categories do not typically allow the developer to add new instance variables, only new instance methods.

Class or Class Object

Primarily, a template for the creation of a group of instances that share common characteristics. A class is also an object in its own right and, as such, has access to class data and methods and can respond to messages.

Delegation

"Subcontracting" the actual work of performing some method out to some other object. From the perspective of an external object, it appears as if one object is performing a particular method when, in fact, that object is having the work done by yet another object.

Dynamic Binding

Another term for late-binding.

Early Binding

Resolution of message references at compile time rather than run time.

Encapsulation

A term describing the indivisible nature of the data and operations that make up an object.

Event-Driven Applications

An application with a flow of control that is determined by a response to certain conditions or (often user generated) events, as opposed to a rigid programming logic structure determined a priori by the developer.

Inheritance

The ability of an object class to obtain some portion of its data and methods by absorbing, in toto, the data and methods of its parent class or classes.

Instance or Instance Object

A specific occurrence of a member of some object class. For example, "John Doe" might be an instance of the class "employee."

Late Binding

Resolution of message references at run-time, rather than compile time. In some cases, systems which support late-binding are, in fact, hybrid systems which resolve references at compile time when possible (for performance reasons) and perform late-binding when necessary.

Method

One of the set of operations that have been defined for some object. Analogous to a function or procedure in conventional programming.

Multiple Inheritance

A characteristic of some (not all) object-oriented environments that allows an object class to inherit from more than one parent class.

Object Hierarchy

A collection of object classes arranged in a parent-child relationship with one another. Classes at the bottom of the hierarchy are more specialized than those above. The object hierarchy is dynamic; that is, it changes over time. The object hierarchy defines the operations possible at any given point in time.

Overloading

Replacing an inherited method with a locally defined method of the same name.

Persistence

The ability of an instance to exist or appear to exist over a scope that exceeds the run-time of any given application. Instances are inherently transient. That is, they cease to exist when the application which created them ceases to run. Persistence is a recognition that, in some cases, the transient nature of objects is undesirable and some method must be provided to give them an existence that exceeds the run-time of any one of the applications which utilize those instances.

Polymorphism

The OO characteristic that different objects can respond to the same message in different ways.

Protocol

A set of methods to which a particular type of object is expected to respond. Often, object-oriented designers will develop a protocol for objects with which their objects or applications can work. These objects and applications should then work with any object class which implements the methods defined in that protocol.

Remote Object

An object which can be accessed over the network by an object running on a physically separate computer.

Root Class

The one class within an object hierarchy which has no parent. Typically, this class is provided by the vendor of whatever OO environment is being used and contains methods and data associated with all of the functionality that is common to all objects within that environment.

Run-Time Binding

Another term for late binding.

Single Inheritance

An object hierarchy in which each class (except for the root class) has exactly one parent.

Static Binding

Another term for early binding.

Subclassing

The creation of a new object class as a child of an existing class. The newly created child class retains all of the method and data definitions of the parent class.

Supering

The ability for a developer to access, within a subclass, the methods defined by a parent class, even if those methods are overridden by that subclass. This is typically used when the developer wishes to retain the original behavior of the class, but wants to augment that behavior in some way.

**DATE
FILMED**

8/12/93

END