

# Audit Considerations of Object-Oriented Information Systems

Donald Golden, (golden@cis.csuohio.edu), Cleveland State University  
Linda Garceau, (garceau@popmail.csuohio.edu), Cleveland State University

## Abstract

*The object-oriented approach has changed the way in which designer/developers build systems. With this approach data and function are linked in a common entity called an object. These objects are reusable and can inherit characteristics from antecedent objects. Such characteristics enable the designer/developer to build a "new" system by reusing objects drawn from a common object library. The characteristics of reusability and inheritance require the designer/developer to "trust" an object. It is the responsibility of the auditor to determine if such "trust" is warranted.*

## Introduction

Reusability and inheritance are among the properties of the object-oriented approach to developing information systems that have resulted in its rapid acceptance by the systems development community. These properties enable developers to focus upon the new problem, building upon prior system solutions, rather reconstructing the old. Yet, they also present a significant control problem for the systems auditor. Reusability means that the contents of previously constructed objects are, by intent, not visible in the current system. Inheritance has the effect that a class may be a many-times descendent of other classes, all of whose data and procedural code is available for use by the current class, but again, not readily visible. Obviously, the integrity and accuracy of processing become primary concerns for the systems auditor with this new approach.

*Readers with comments or questions are encouraged to contact the authors via e-mail.*

## The Object-Oriented Approach

In the late 1960's and early 1970's, Stevens, Constantine, Myers, Yourdon, and others [1, 2] proposed the systems analysis and design technique known as structured analysis and design. Widely accepted, this approach emphasized the hierarchical organization of the system and the functions that were required to solve problems. Yet, other system professionals criticized this approach's failure to place adequate emphasis upon data. Data structures that are used in a problem solution are only loosely coupled with the procedures that act upon them. Over time, it became obvious that the role of data in the development process was much more important than was first realized. In the mid-1980's, an alternative method which linked data and function together began to attract more and more attention. This approach was called "object-oriented".

The object-oriented approach, unlike the structured approach, combines data and function in a single entity called an object. Formally defined, an object is an individual, identifiable item, unit, or entity either real or abstract, with a well-defined role in the problem domain. Objects can represent tangible things such as, houses, cats or accounts; roles played by an individual, such as father, teacher, or customer; incidents, such as performances, events or games; interactions or relations between objects, such as classes or purchases; and specifications that have the quality of a standard or definition, such as car model or policy type.

In the object-oriented approach, objects are viewed as “black boxes.” They *encapsulate* both data and function, with their internal workings hidden from their users. Common to every object is a *state* and *behavior*.

An object’s state is determined by the static properties or attributes that are unique to that object. As the values of these attributes change, so does the object’s state. For example, in an object known as account such attributes might include account number, name, address, and balance; while in an object known as game, they may be date, teams competing, and score. With the object-oriented approach, the state of an object usually is private, not being visible to other objects. This denial of access to the internals of an object is known as *information hiding*, and it has the advantage that the data can only be manipulated in ways permitted by the object.

The behavior of an object is the set of operations or responsibilities that the object must fulfill either for itself or for other objects. Thus, an object may act independently or be called upon by other objects to act for them. When these actions are performed by the object, they will change the state of the object. For example, an account object may be called upon by a purchase object to alter the balance of the account, or a student object may recalculate GPA when called upon by a grade object.

Once the object has performed the action, the state of the object, which was altered by the action, remains unchanged until the object is called upon again. Like the states of the object, some of the responsibilities or actions of the object are also hidden. Thus, they are known only to the object. Those actions or responsibilities that are visible and known outside of the object form the interface of the object, defining how it will act and react with other objects in the problem domain. Theoretically, the integrity of an object can not be violated, since it can only behave and interact with other objects in a well-defined ways.

Implied in this discussion of state and behavior is the concept of collaboration. Because objects can not always carry out their responsibilities independently, they must rely upon other objects for assistance. Such help is called collaboration, and the objects relying upon one another, collaborators. For example, an account object may collaborate with a deposit object to calculate a new account balance, and a student object may rely upon a grade object to recompute a student’s overall GPA. Through collaborations, objects work together to achieve the overall goals of the system.

A final important component of the object-oriented approach is *class*. While collaboration indicates how objects work together explicitly to perform a task, class identifies the implicit relationship of objects. Every object is a member of a class. Such membership identifies the essential attributes and behaviors of objects that are shared by every member of the class. Another way of looking at the class/object relationship is to say that a class is a model, or template, for creating an object, and every object is an instance of a class.

For example, in a sales system there is a class called “customer.” If there are several basic types of customers, the customer class may have *descendent* classes such as “product customer” and “service customer”. Attributes common to all types of customers, such as ac-

count number, name, street address, and balance; and common behaviors, such as “make sale” and “collect moneys,” are defined in the class customer, and are present not only in customer but in its descendent classes (product customer and service customer), as well. We say that product customer and service customer *inherit* the attributes of their parent class, customer. Only attributes and behaviors that are unique to a descendent are defined at the descendent level. Such unique attributes might be “warranty terms” for the product customer class or “service tax” for the service customer class; unique behaviors might include “identify common carrier” in the product customer class or “identify service personnel” in the service customer class.

The relationship between classes and objects is analogous to the relationship between data types and variables in older programming languages. You can have variables of type “integer”, and you can define variables of this type called `Account_Number` or `Number_of_Customers`. Similarly, you can have a class called `ServiceCustomer`, and define objects of class `ServiceCustomer` called `New_Customer` or `Current_Customer`.

### **The Power of Objects**

The object-oriented approach derives its power from several factors. First, the fact that objects model components in the real-world system makes it relatively easy to reuse the objects when new programs are written for a system. Indeed, if the classes that describe a system are well designed, building new programs from objects of these classes tends to be much easier than using older techniques. Second, the ability to create descendent classes that inherit data and procedural characteristics from their parents makes it easier to adapt existing classes to meet new conditions. The ability to reuse objects permits the system developer to “write it once,” and then replicate the objects in all projects requiring identical functionality. Like a building block, a good object is designed and written us-

ing techniques that enhance its purpose, strength, and form. Subsequent developers are not concerned with how an object performs its function, but rather what the function is. By combining these “black box-like” objects, system developer can reduce the costs of building the system and minimize the time required by the process, while ensuring the delivery of a relatively sophisticated product .

### **Building an Object-Oriented System**

Developing an object-oriented system is much like building a house. An architect designs the house assuming that certain standard components will be used. For example, vertical wall supports are 2x4 inch boards and the separation between them is 16 inches. Doors and windows are selected from standard sizes and shapes. Bricks or concrete blocks have standard sizes and structural properties. When a mason selects concrete blocks from a warehouse, he knows the properties of these blocks because they were all made to match a particular specification. All building blocks of a particular type are the same. If the worker must modify a block to fit a certain purpose, he alters only that single block. He does not cut into the block to ascertain its composition or strength; because he is confident that it is suitable for his purposes.

For the mason, the confidence in the quality of the block is of utmost importance. He must have confidence in the original design and composition of the block. He also must have confidence in the process by which it is made. If either product or process are deficient, the wall that is built by the mason will also be of poor quality, no matter how well he does his job. In addition, he must use the proper block for the proper job. If, for example, he uses a ceramic block intended for decorative use to build a load-bearing wall the result will be disastrous, in spite of the fact that the block performs exactly as its specifications required.

Similarly, a systems designer/developer must have confidence in the quality of the objects

with which he builds his system, and must know how the designers of these objects intended them to be used. Such confidence comes with knowledge of the design/development process and product. If either process or product is deficient, then the system built with them will also be deficient.

### **The Risks of the Object-oriented Approach**

Reusability of code and inheritance of characteristics and behaviors can minimize system investments. Yet, successful system development utilizing the object-oriented approach demands that object specification is done right the first time. The development of deficient objects that are reused, or whose behaviors and characteristics are inherited by other objects, presents a significant problem. This problem is exacerbated by the fact that objects function like black boxes. Using this approach, a programmer may not have an opportunity to “peak inside” and audit the object before it is used in the system. Instead, when he takes objects off the shelf to use in his design, he must rely upon the specifications of those objects and the accuracy and integrity of their construction. Moreover, since the objects the programmer uses are quite likely themselves to be constructed using other objects, the chain of dependence can stretch back almost without limit.

### **Minimizing the Risks**

While it is possible to reduce the risks involved with object-oriented systems, it is extremely difficult to eliminate them entirely without also eliminating the advantages of object-oriented systems. As with most business endeavors, one must evaluate the benefits to be gained from increased control and weigh them against the costs of implementing the necessary security measures.

To begin with, one must divide object-oriented languages into two categories: those that allow the user to develop new classes, and those that simply provide a library of useful ob-

jects with which to build applications. Languages in the first category include C++ and Smalltalk, while languages in the second include Visual Basic and Microsoft Access. Clearly, the first category is more likely to be used by professional programmers and subject to traditional development and change controls. The second category is more likely to be used by end users. Since it is not possible to cover both categories in a single paper, the following discussion is limited to the second category of languages, using Microsoft Access and Visual Basic as examples.

Systems developed using object-oriented techniques are subject to the same risks that have always existed in computer software, particularly end-user systems which frequently are developed by people who are not aware of the potential problems. However, object-oriented systems present additional risks, particularly from the point of view of the auditor, because much of the system specification is not visible. For example, an Access database can allow one to embed an Excel spreadsheet as a field in a table using Object Linking and Embedding (OLE). The spreadsheet is not actually stored in the database. Instead, a dynamic link is created from the database to the spreadsheet. The advantage of this approach is that if the spreadsheet is updated, the database always shows the most current data. Unfortunately, this also means that in order to audit the database it is also necessary to audit the spreadsheet. Indeed, the spreadsheet can contain embedded Word documents, which can, in turn, contain other embedded databases.

### **Identifying Well-Controlled Sources**

To minimize such risks, users must identify a well-controlled source of “starting objects.” Such object sources can include vendors such as Microsoft or Borland, or other reliable, independent software providers. While there obviously is no guarantee that software produced by firms such as these is foolproof, for most practical purposes, one can assume that their objects are error-free. At least, they are more

reliable than objects such as Visual Basic controls (VBX files) written by an associate down the hall.

When defining a set of starting objects, one should also consider whether or not to permit the use of objects that utilize a network, particularly one that might go outside the organization. The basic function of an object is to perform a task without requiring that the application developer know how that task is performed. If the task is to retrieve data, it may not be clear that the object actually is retrieving the data via a network from an uncontrolled source. In turn, it may not be apparent to either the developer or the auditor that the object can also update a database in a way that bypasses normal control procedures. Admittedly, these examples represent extreme (and unlikely) situations, but without access to the object's source code it is not possible to guarantee network control.

From the starting set of controlled objects, the next step is to use these objects to create applications, a task which usually involves creating new objects. For example, Microsoft Access treats forms, reports, queries, and even Wizards as objects which can be stored in a library database for use in other applications or by other application developers. A useful tool for risk management in the object environment is the object library catalog (OLC).

### The Object Library Catalog

The object library catalog (OLC) is an inventory of all audited objects in use in the organization. Specifically, it includes the name of the object, its owner(s), a description of the object's function, a listing of those systems which reference it, the dates upon which the object were modified and by whom, and a copy of the code, if available. The purpose of the OLC is to identify for the developer and the auditor the object's source and the magnitude of its risk.

The object library catalog identifies the source of risk first by identifying the objects in

use. If an object is not identified in the OLC, it should be considered as a potential threat to the accuracy, integrity and security of system processing. The presence of an object in the OLC provides *limited* assurance that system management and auditors are aware of its existence and have evaluated its processing.

The object library catalog provides additional information to the auditor in the assessment of an object's risk. The magnitude of this risk is a result of three factors: the function of the object, the extent of the object's use, and the object's development/maintenance history. Each of these factors is defined in the object library catalog.

The function of the object relates to its business or systems purpose and appears in its description in the OLC. From this description, the auditor may evaluate its risk. For example, an object whose function is to execute a significant business calculation (e.g., amortization of debt repayments) would have associated with it a high degree of risk. On the other hand, an object whose function is to provide a choice of menu options would be identified as having low risk. Risk is also magnified by the extent of the object's use. An object that is referenced by only a limited number of systems within a single user area would have low risk, since the impact of an intentional or unintentional error is isolated. Objects, however, that are used by multiple systems in multiple areas would have significantly higher risk.

Documentation of development and change history provides additional assurance as to the integrity, accuracy and security of the code. Unmodified objects, developed by external vendors, would generally be considered reliable. Similarly, objects developed and maintained by the organization's information systems staff would have greater reliability than those developed and maintained by end users. Auditors also should be concerned with last change date, particularly, if the object had been reviewed during the course of their last audit. If

the object has undergone review and remains unchanged, its reliability may be further assured.

**Controlling Object-Oriented Risks**

The unique risks of object-oriented systems fall into two basic categories: (1) because objects are designed to be used as black boxes, it frequently is not possible to see the internals of an object or to determine what other objects it may use; (2) because of the ability of object-oriented systems to link to other objects dynamically during execution, it may not be possible to tell which objects a system accesses just by looking at the system's code.

The first type of risk is controlled through an iterative process. If an object was developed by an approved vendor or is listed in the OLC, the only additional requirement is to ensure that it is used within the limits of its specifications. If so, it can be assumed that the object adds no new risks to the system.

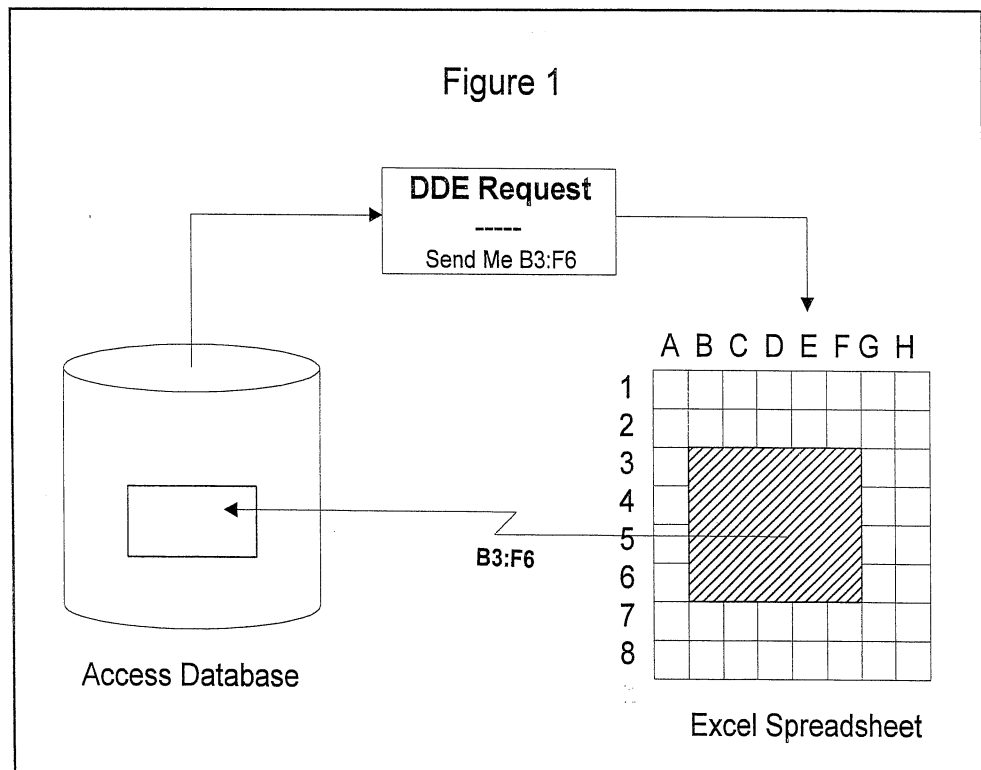
If the object is being developed as a new object, it is necessary to verify that it consists only of low-risk objects that are used in ways that do not add a risk to the system. For example, an Access object that uses a restricted database should verify that the user is authorized. Once this has been done, the new object can be considered low-risk and can be added to the OLC. The result of this approach is that it is never necessary to look at more than the specifications of the current object to de-

termine its risk level. If it consists entirely of low-risk objects, it is irrelevant how those objects are implemented.

Controlling the second type of risk, dynamic access of objects, involves two factors: (1) be aware of what actions or object properties will create a dynamic link; (2) if at all possible, restrict the user's ability to access objects outside the current system. Although an auditor clearly does not need to understand the programming details behind dynamic access, he should understand the basic concept and be able to identify the statements that indicate that dynamic access is being used.

Software designed to run under Microsoft Windows can access data and code dynamically using three techniques: (1) dynamic data exchange (DDE); (2) dynamic link libraries (DLL); and (3) object linking and embedding (OLE).

DDE was developed as a mechanism for implementing client-server activities. For example, consider the situation shown in Figure 1.

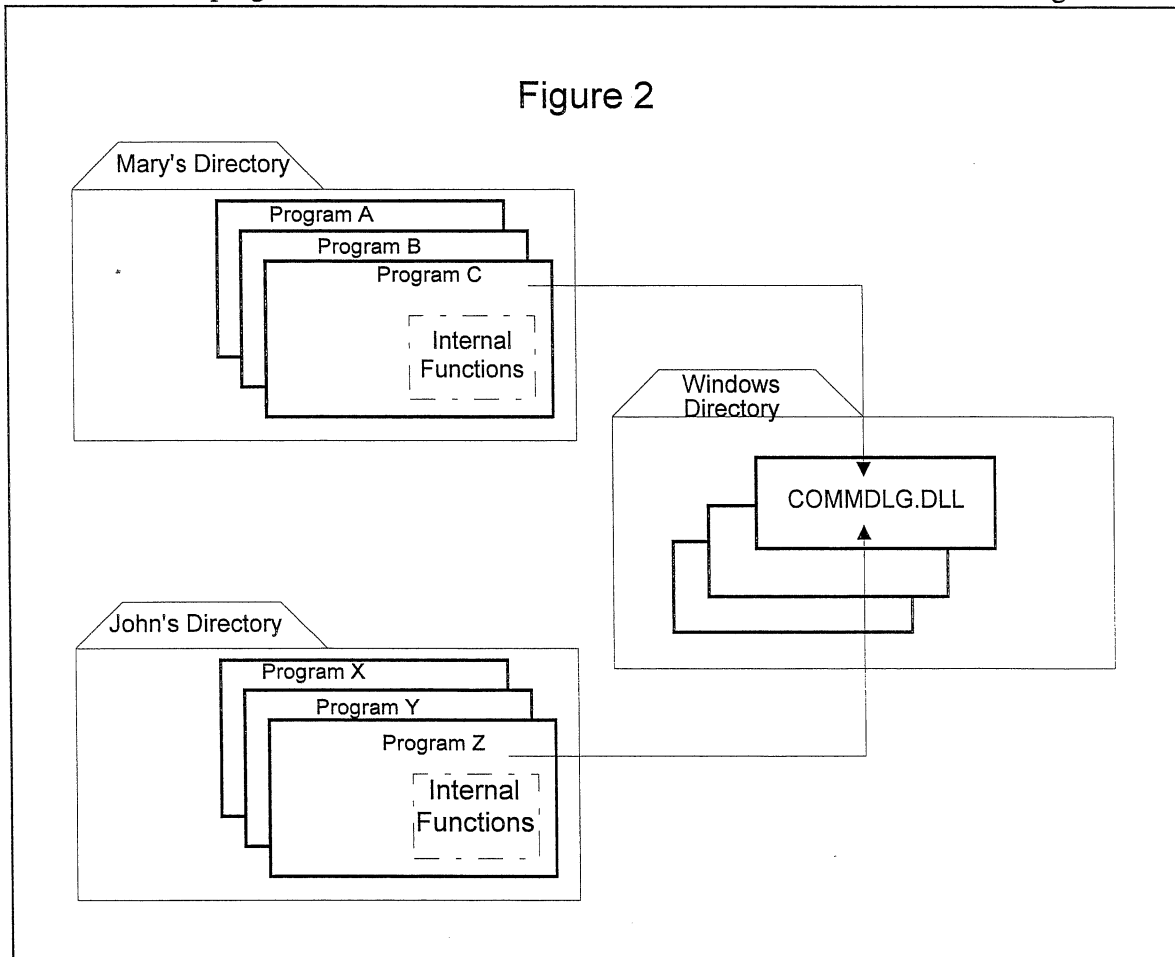


An Access database needs a piece of data which is calculated by an Excel spreadsheet. The database initiates execution of the spreadsheet, requests that a particular range of cells be copied into the database, then closes the spreadsheet. This means that for purposes of risk control, the spreadsheet is part of the database system. The functions in Microsoft Access that control the use of DDE all begin with the letters "DDE." In Visual Basic a DDE connection is called a *link*, and the control properties and events related to DDE processing all begin with the word "Link." If any of these key words or phrases are used in a system's code or in a control's properties, the system is using DDE and the system or document which is being connected to the current system should be identified and audited.

time rather than during compilation. The reasons for this approach are important: a DLL can be upgraded without having to change or recompile the program that uses the DLL, and a DLL can be used by many different programs at once. As a result, Windows software uses DLLs extensively. For example, in most Windows application software, if you select file processing options such as OPEN, SAVE, or BROWSE, the program opens a standard window (called the *common dialog* window) that allows you to search for the directory or file that you want. The program code that drives this function is contained in a DLL (commdlg.dll) located in the Windows directory, and is shared by virtually all Windows software. Figure 2 shows the relationship between the common dialog DLL, located in the windows directory, and two programs that use it, each located in a different user directory.

A DLL is a library of functions and procedures to which a program is connected at run

It should be clear that although DLLs are



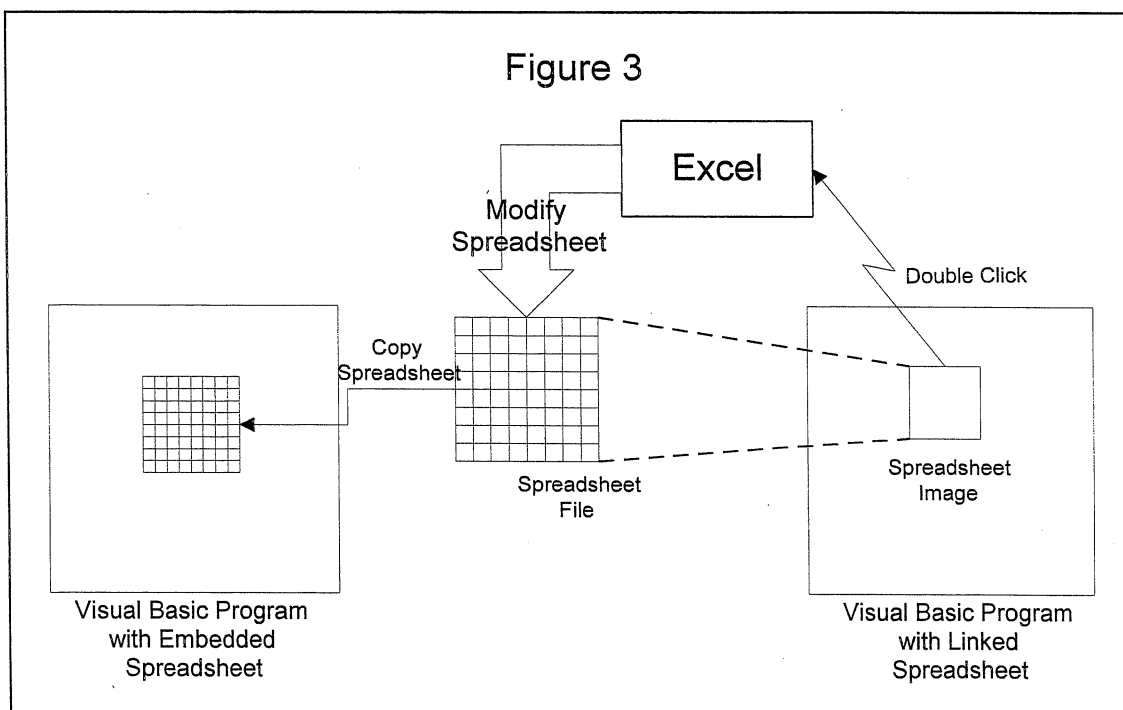
very powerful, useful, and almost impossible to avoid, they also make it extremely difficult to determine exactly what code a program executes. If the only DLLs that a system uses are those provided by a reliable vendor the risk is very low, although it is difficult to be certain that the original DLL has not been replaced. However, a program can also call user-written DLLs, which generate a much higher level of risk and requires that the DLL be audited as part of the system. In Visual Basic or Microsoft Access, one can tell if a system uses a user-written DLL by looking for a DECLARE statement that contains a LIB clause. This statement identifies the subroutine being used and the name of the DLL that contains it.

The third technique is object linking and embedding (OLE). There are two different types of OLE objects: linked objects and embedded objects. Both allow a system to connect to an external object, even one developed by a different application. For example, an OLE object placed in a Visual Basic form can connect to an Excel spreadsheet. The difference between linking and embedding relates to where the object's data is stored. In the previous example, if the spreadsheet is *embedded* in the form the

spreadsheet data is stored in the Visual Basic application. Changes to the data can only be made through the Visual Basic program and they do not affect the original file. If the spreadsheet is *linked* to the form, a connection is made to the Excel file and no data is stored in the Visual Basic system. Figure 3 demonstrates the two configurations.

When the Visual Basic system is running, if the user double-clicks on the image of the linked spreadsheet, Excel is activated and the spreadsheet can be manipulated as usual in Excel. Furthermore, any changes made to the spreadsheet outside the Visual Basic system, using Excel, will be reflected in the linked object the next time the Visual Basic system runs. With the embedded spreadsheet, a copy is inserted in the Visual Basic object but there is no further connection to the original spreadsheet or to Excel.

In order for an object-oriented application to use OLE, it must contain either an OLE control (Visual Basic) or an object frame (Access). In Access an OLE object can also be created and manipulated during execution of a program. This requires a "Dim...Object" state-





ment to declare the object, and either a "CreateObject" statement or a "GetObject" statement to use the object. In Visual Basic the OLE control must be defined as part of a form, although it may be invisible when the application is running. Statements that manipulate the control during execution use OLE properties such as Action, Autoactivate, SourceDoc, SourceItem, or properties whose names begin with "Object" or "OLE".

The first step in controlling these risks requires restricting the ways in which object-oriented programs can work. For example, if a Visual Basic system creates a dynamic link to another system such as an Excel spreadsheet, do not permit the user to supply an arbitrary object name (i.e., the name of the spreadsheet) at run time. Instead, provide a list of acceptable Excel files from which the user may select. This principle should be applied to *any* external data access, whether via a dynamic link to a database, reading or writing a file, accessing a network, etc. A list of permitted data sources should be coded into the program or stored in a secure database and the user required to select from this list. If it is not possible to provide a list of data sources, the system should at least prohibit access to restricted or unsecure sources.

### **Auditing An Object-Oriented System**

The first step when auditing any system is to define the scope of the audit. A valuable tool in this process can be the object library catalog (OLC). The existence of an object in the catalog can enable the auditor to make some assessment of its risk. The omission of an object from the catalog may also identify an audit candidate.

Using an OLC entry (which includes a description of the object's function, a listing of the systems which reference it, and the dates upon which the objects was modified and by whom) the auditor may judge the level of risk. Audit candidate objects are those which either perform significant business function or are

linked to numerous applications. Auditors may choose to test these objects during an audit, especially if they were recently implemented or modified.

Another element of the OLC entry is a copy of the object's source code. To audit a computer information system thoroughly, it is necessary to examine the program code with which the system is implemented. Although validating a system by testing its operation can indicate that the system behaves properly in normal use, it cannot determine what the system will do under abnormal conditions. For virtually all application-level object-oriented languages, this means that it is necessary to print the application using whatever facilities the programming language provides, to show both the code used to create the application and the properties of the objects used. One cannot verify the correctness of a typical object-oriented application system simply by viewing the code for individual objects (controls) on the screen. Since the code usually is presented in alphabetical order according to the names of the controlling objects, simply *finding* all the code is not an easy task. Furthermore, the values assigned to properties of objects can have a significant effect on the operation of a system. For example, in Visual Basic the DataField and DataSource properties control database access, while the various Link properties can create a run-time link between a control and other applications.

Printing the *basic* program information for typical object-oriented application languages usually is not a problem. In Visual Basic, for example, if the program files are saved as text files they can be displayed or printed with any ASCII text processor. Alternatively, the Visual Basic PRINT function gives one a choice of printing selected forms or all forms. For each form, one can print an image of the form; the code that defines the form, all its controls, and all their properties; and all procedural code for the form and its controls. In Access, the file menu provides a PRINT DEFINITION function which prints the basic table definitions, and Mi-

Microsoft provides a database documentor as an add-in package.

Unfortunately, the program documentation capabilities provided as part of these object-oriented application languages are fairly basic, and program analysis capabilities are nonexistent. An auditor who works regularly with sophisticated applications developed in a few specific languages should seriously consider using third-party audit support software. The reports produced by these packages usually are more extensive and more flexible than those provided by vendors.

For example, a common problem is to determine whether or not the current version of an Access database system is the same as a base version. Because Access stores both data and objects in a single file one cannot simply compare the files, since any change in the data would cause the files to be different. However, a typical third-party audit package will compare two databases and produce reports describing any differences between them. The package should have options to specify which objects to compare, whether or not the data also should be compared, and which reports should be printed. Commonly, such a package will also allow one to compare similar objects in a single database. In choosing such an audit support tool, it is also useful to select a tool that will analyze the structure of a database, showing such characteristics as relationships between tables, relationships between objects, cross references between objects, cross references between subroutines, formatted listings of subroutine code, properties of objects, and so on.

### **Maintaining Control**

Once an information system has been audited and certified to be acceptable, it is critical to ensure that the system will not be altered by unauthorized personnel. The precise technique to be used here depends on the software being used and the facilities available. For example, Visual Basic provides little in the way of

security locks, but it does allow the developer to compile the final system into an executable ".EXE" module which cannot be altered by most programmers.

On the other hand, Microsoft Access provides extensive security locks on all objects in a system, from the data through reports, forms, etc., and can be set to restrict access to specific groups or specific individuals. These security features not only control access to a specific application, they control the ability of programmers to use objects developed for one system in a different system, or to link to a database through an OLE connection. Software that is to be made available for general use can be placed in a special type of Access database called a library and locked to be made read-only for unauthorized personnel.

The most difficult part of the system to control is the dynamic link libraries. Since the link between the calling program and the library is made during execution, there is no way to guarantee that the DLL has not been changed since it was audited. Nonetheless, there are some actions that can be taken to reduce the risk.

In the standard, single-user DOS or Windows environment there is no way to lock a file or directory to protect it from change. In this situation the only thing that can be done is to identify all DLLs used by a system, then compare them to base versions of the same files. In a network environment there usually are ways in which files can be secured against unauthorized change, especially if they are located on a separate file server; the exact technique and the degree of sophistication vary with the network software being used. In the non-network environment, however, the best solution again lies in third-party software. Software security packages can be obtained for very nominal prices, and can restrict the ability of an unauthorized user to change, read, or even see specific files or directories.

Perhaps the final question to ask regard-


ing software control is: how much is enough? To begin with, it is impossible to prevent a highly motivated, highly skilled expert from accessing software or data. What one person can lock someone else can unlock. What one *can* do is to make it as difficult as possible for the unauthorized user. If nothing else, this greatly increases the chances that an intruder will be caught. However, it also makes it more difficult for authorized system users to do the work they need to do, and for programmers to maintain or improve the system. In general, the more extensive the controls on a system the more difficult it is for legitimate users to use the system. This certainly does not mean that control over object-oriented software should be ignored. However, the degree of control and security should be proportionate to the value of the system and not installed simply because it can be done.

### Conclusion

The same properties—reusability and inheritance—that have resulted in the rapid acceptance of object-oriented languages and methods present auditors with a formidable challenge. These properties, which support the accelerated development of new applications, assume the reliability of the objects used in the construction of the system. It is the responsibility of the systems auditor to put in place procedural and system controls that ensure that objects meet the developers' functional expectations. Subsequently, the systems auditor must review and test the controls and code to ensure that they have not been later compromised. In this new environment, controls are of the utmost importance because any new object oriented system is only as strong as its weakest object.

### Implications for Future Research

This discussion has identified some of the weaknesses inherent in the object-oriented approach to systems design and development. Using examples drawn from languages such as Microsoft Access and Visual Basic, it has attempted to recognize the shortcomings that may

affect system integrity and present a substantial risk to the auditor. Significant research opportunities remain open to investigation. Researchers are encouraged to evaluate from a technical perspective the risks inherent in other object-oriented languages (C++ and Smalltalk), identifying language specific functions that can be used to minimize such risks. It is also necessary to assess the level of awareness within the audit community as to the magnitude of risk presented by object-oriented design approaches and applications. As older systems are replaced by object-oriented applications, these risks will become more evident. In turn, auditors will have to enhance their technical understanding of object-oriented languages and their capabilities. Finally, it is imperative that researchers develop an inventory of hardware and software-based tools that can ensure the integrity of objects in an intranet and Internet environment. 

### References

1. W. P. Stevens, G.J. Meyers, L.L. Constantine, "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2, pp. 115-139, 1974.
2. E. Yourdon and L.L. Constantine, *Structured Design*, Prentice Hall, Englewood Cliffs, N.J., 1979.

