Internal Control Methods In Object-Oriented Systems

P. C. Chu, (Chu.4@osu.edu), Ohio State University

Abstract

Object-oriented programming languages offers many advantages over the traditional programming languages. It is expected that object-oriented technology will become the predominant programming paradigm in business. However, as object-oriented technology incurs a fundamental shift in programming paradigm, new internal control risks as well as opportunities will arise. It is imperative that these risks be understood and that effective control methods be developed to contain the risks before object-oriented systems proliferate in business organizations. It is also important to take advantage of the power of object-oriented technology in implementing internal controls. This paper analyzes object-oriented programming, studies the ensuing internal control risks and opportunities, and develops control methods to contain the risks.

1. Introduction

Business firms have relied on third-generation programming languages, mainly COBOL and FORTRAN, to construct their mission-critical software. One of the shortcomings of these languages is that they are not geared toward representing the real world. Rather, they provide constructs that largely reflect the physical and internal operation of computers. Developing systems to solve real-world problems with these languages proves to be highly procedural (lengthy and laborious), costly, and error-prone.

In recent years, object-oriented programming is rapidly emerging as a new and superior paradigm for developing computer systems. The superiority of object-oriented technology stems from a sophisticated modeling capability and innovation in code re-use. These features bring about a substantial reduction in the procedural nature of programming and a corre-

sponding gain in efficiency, reliability, and ease in the system development process.

Object-oriented technology, however, incurs a fundamental shift in programming paradigm. It is so different from the traditional programming tools that its deployment in organizations is bound to create new internal control risks and opportunities. Hitherto, object-oriented technology remains largely unknown to accounting students, scholars, and professionals. Research in object-oriented accounting systems has barely begun. The specific issue of how to control object-oriented technology has not been investigated.

It is imperative that IS (information systems) auditors understand this new technology and its control implications. In particular, there is an urgency to develop effective preventive control methods before object-oriented systems proliferate in business organizations. It is the objective of this paper to study the internal control risks and opportunities arising from the de-

Comments and Questions concerning this article should be sent directly to the authors via e-mail.

ployment of object-oriented technology and to design internal control methods to contain the risks.

Object-oriented technology has made progress along two paths: programming languages and databases. Of the two, object-oriented programming languages are the more developed. Large-scale systems have been and are being built with object-oriented languages. By comparison, object-oriented database systems are still in their infancy and remain largely experimental at this time. The discussion of this paper is mainly confined to object-oriented programming languages. The analysis focuses on the conceptual and theoretical levels rather than on implementation details.

The remainder of this paper is organized as follows. Section 2 presents core object-oriented concepts with examples. Differences between the object-oriented programming paradigm and the traditional programming practice are analyzed, which paves the way for the conceptualization of risks and opportunities brought about by the new technology. Section 3 reviews previous research in object-oriented accounting systems and security models in object-oriented systems. Section 4 examines internal control risks and opportunities resulting from the distinct features of object-oriented technology. In Section 5, methods for controlling object-oriented technology are developed. The focus is on preventive control measures. Section 6 concludes the paper. Section 7 suggests some future research directions.

2. Object-Oriented Paradigm

Object-oriented concepts are the products of a long process of development and research in three disciplines: programming languages, artificial intelligence, and databases. Simula-67 (Birtwistle et al., 1973), which provides the fundamental notions of objects, messages, and classes, is generally considered as the first object-oriented programming language. Since Simula, the development of object-oriented programming languages has followed two different

paths. One was the development of completely new object-oriented languages notably Smalltalk (Goldberg and Robson, 1989). The other was to extend conventional programming languages to include object-oriented concepts, e.g., Objective C and C++ as extensions of C language. In the field of artificial intelligence, Minsky (1975) originated the idea of frames as a knowledge representation scheme. Frames provide the concepts of inheritance and behavior encapsulation. In the database area, developing data models to capture more semantics than the relational data model has always been an important research endeavor (Abrial, 1974; Bretl et al., 1989; Hammer and McLeod, 1981; Smith and Smith, 1977a, 1977b). This effort generated the ideas of aggregation, generalization, and instance-of relationships that parallel similar concepts developed in programming languages and knowledge representation schemes.

Because of the diverse paths along which object-oriented concepts have been developed, there is still a lack of a single standard for object-oriented concepts (Kim, 1990). However, a set of core object-oriented concepts common to most object-oriented programming languages and database systems has emerged (Kim, 1990). This section briefly presents these concepts and analyzes the differences between the traditional programming languages and object-oriented programming languages. These differences underlie the need for new internal control strategies and practices.

Core Object-Oriented Concepts

The core object-oriented concepts are embodied in the following terms: object, attributes, methods, encapsulation, message passing, class, class hierarchy, inheritance, and polymorphism.

Object

In the object-oriented approach, any entity whether it is a concrete object, a concept, or a classification is uniformly modeled as an object. For example, a checking account main-

tained at, say, the First National Bank, the general ledger account CASH, or an asset classification such as CURRENT ASSETS, can all be modeled as objects. Furthermore, each object is identified by a unique object identifier (OID) rather than by attribute values.

Encapsulation and Message Passing

Every object has a state and a behavior. The state of an object is represented by its attribute values. The domain of an attribute may be either atomic or arbitrarily defined. Because the domain of an attribute may be arbitrarily defined, an object may take on a complex structure. That is, some or all of the attributes of an object may have their own set of attributes. The value of an attribute of an object is also an object. Thus, the state of an object is actually represented by a set of identifiers which are the values of the attributes of the object.

The behavior of an object is the set of methods or programming procedures that the object is capable of executing. Like its state, the methods (behavior) of an object are *encapsulated* within the object. They are evoked from the outside through sending the object a message, which has the following form: "**Receiver selector**". Receiver is the object to which the message is sent. Selector is the name of the method that the object is to execute. A method may have zero, one or more arguments.

As an example, an object representing a checking account may have an attribute *balance* to represent its current balance, a method called *debit* to increase its balance, a method called *credit* to decrease its balance, a method called *reportBalance* to report its current balance, etc. The following message will cause the checking account maintained at the First National Bank to increase its balance by \$100:

"CheckingAccountFirstNationalBank debit:100",

CheckingAccountFirstNationalBank, which is the receiver of the message, is an object created to represent the checking account a firm maintains

at the First National Bank. The selector is "debit:", which takes one parameter in this simplified example.

Class

Objects that share the same set of attributes and methods are organized into classes. A class is a means of grouping objects. An object must belong to only one class as an instance of that class. The relationship between an object and its class is the INSTANCE-OF relationship.

A class has two roles to play. First, it defines the structure of the instances of that class by identifying the set of attributes and methods shared by all the instances of that class. Mylopoulos et al. (1980) called this function the "definitional" role of a class. Second, a class is an object in its own right. It may have its own attributes which describe the properties and facts of the class taken as a whole. The term "class attributes" is used to differentiate the attributes belonging to a class from "member attributes" (Hammer and McLeod, 1981) or "instance attributes" (Goldberg and Robson, 1989) which belong to the instances of a class.

Figure 1 illustrates the INSTANCE-OF relationship between the CASH class and its three instances. Instance attributes and instance methods can be defined that will be used to store and manipulate data in the three instances or any new instances to be created in the future. For example, an instance attribute *balance* can be defined which will be used by each instance to keep track of its current balance. An instance method *debit:* can be coded that, upon execution, will increase *balance* by a certain amount. In addition, it is possible to define class attributes and class methods for CASH itself. For example, CASH may have a class method *report-total-cash* to compute and report the total amount of cash available from its instances (i.e., the sum of balances in the three checking accounts).

Class Hierarchy and Inheritance

Classes that share common features can

Figure 1
Illustration of Class and Instance-of Relationship

form a class hierarchy. A new class may be specified by declaring an existing class as its super-class. The new class is called a sub-class of the existing class. A super-class may have any number of sub-classes. A sub-class inherits all the instance methods and attributes of its super-class. It is also possible to specify additional attributes and methods for the sub-class. Some systems such as C++ release 2.0 support multiple inheritance, which allows an object of a class to inherit attributes and methods from two or more super-classes.

In a class hierarchy, the super-class is a general form of its sub-classes. Therefore, class hierarchy captures the generalization (IS-A) relationship between one class and a set of classes specialized from it.

Accountants organize accounts into hierarchies. Figure 2 illustrates the hierarchy of asset accounts. Note that under assets are current assets, fixed assets, and other assets. These are special forms of assets. Under current assets are cash, accounts receivable, marketable securities, and inventory. These are special forms of current assets. Under inventory are raw materials, work-in-process, and finished goods. These are

special forms of inventory. So, a series of generalization-specialization (IS-A) relationships link the account together. Such a class hierarchy can be easily modeled in an object-oriented approach. Note that an IS-A relationship is represented by a solid line in Figure 2.

After a class hierarchy is defined, it is possible to make use of inheritance. For example, all assets have a dollar balance that represents their valuation, which can be defined as an instance attribute *balance* at the top

node of the asset hierarchy. Once defined, this attribute automatically becomes an instance attribute for all the sub-classes of ASSETS.

Polymorphism

Polymorphism allows different objects to respond to the same message with their own behavior. For example, both a square and a circle can encapsulate a method called "compute-area" and thus can respond to the same message "compute-area". However, "compute-area" will be defined differently for a square than for a circle. Each will use its own method to compute its area. Because a method is associated with an object, there is no confusion as to which algorithm to perform. Polymorphism permits semantically equivalent but algorithmically different procedures to have the same name. It also provides a means for objects in the sub-class to over-ride a method specified in its super-class.

Polymorphism allows a natural modeling of the double-entry accounting system. For example, *debit* and *credit* can be defined differently for the two hierarchies of account categories, just as accountants do. In the hierarchy of asset accounts (see Figure 2), *debit* is defined as a

Figure 2
The Hierarchy of Asset Accounts

method that increases a balance, *credit* as a method that decreases a balance. In the hierarchy of liability accounts, the definitions of the two methods are reversed. Once these methods are defined, the recording of a transaction is effected by executing two messages:

Account debit: amount

Account credit: amount

Debit (credit) increases or decreases an account depending on whether Account (receiver of message) is an asset account or a liability account. This way of recording transactions exactly matches the double-entry system.

Differences between Object-Oriented Programming Paradigm and Traditional Programming Practices

The above discussion reveals a number of differences between an object-oriented model and the traditional programming practice. Their main difference lies in the programming paradigm. There are also differences in program-

ming environment, structural representation, semantic expressiveness, and polymorphism.

Programming Paradigm

Data and programs are the two key ingredients of computer systems. In the traditional programming practice, they are kept separate. During processing, data, which usually are packed into records and files, are accessed and manipulated by program instructions. After processing, they are re-stored. Program instructions are based on the constructs of a programming language. The constructs of third-generation programming

languages consist mainly of hardware-oriented commands such as moving data within the primary storage area, assigning values to memory addresses, making arithmetic computations, storing and retrieving data to and from external memory, etc. This mode of operation has a number of characteristics. First, programming is dictated by internal working of computers. Consequently, there is a lack of naturalness in the modeling process. Second, meaningful operations of computers are few in number. Hence, the vocabulary (the set of constructs or commands) of a third-generation language, which models the operation of computers, is also very small in size. As a result, each program statement advances only marginally toward the final solution and a program usually consists of a large number of lines. Problem-solving logic is obscured. This phenomenon, which is described by the term "procedurality", creates great difficulty in program construction and maintenance. Third, owing to the separation of programming statements and data structure, code sharing across programs is difficult.

Object-oriented programming has a completely different orientation. First of all, data and program instructions are integrated into objects. A computer program consists of a sequence of messages requesting different objects to perform the functions they are designed to perform. Processing is mainly carried out by objects drawing on their methods and data store. Since objects model real-world entities, their methods, instead of representing computers' inner operations, are related to applications. Thus, the orientation is shifted from hardware to applications. As a result, object-oriented programming is more natural and intuitive, and, hence, more appealing than the traditional programming practice. Second, the vocabulary of object-oriented programming languages is the set of methods defined for all objects. Since there is no limit to the creation of classes and methods, the vocabulary of object-oriented programming languages is virtually limitless. Herein lies a potential for drastically reducing the procedural nature of third-generation programming languages and thereby for solving the many problems caused by it. Third, objects, in providing both a structure

tremendous gain in productivity is possible.

Programming environment

Another important difference between the traditional programming practice and an object-oriented approach is the programming environment. In the traditional practice, the scope of a program is well defined and stable. The boundary of a program is delineated by its own code and whatever external procedures called. The little sharing of code among programs largely insulates a program from changes in other programs.

In contrast, the environment for an object-oriented program is the totality of classes and objects existing in the entire enterprise system (as opposed to a single application program) with all of their methods and data. Such an environment is called an "image" in Smalltalk. An object can potentially interact with and draw upon all objects and classes that exist in an image. This all inclusive nature of an object-oriented model provides the necessary foundation

Figure 3
Dynamism in object-oriented Technology

to store data and methods to carry out actions, are self-sufficient and serve as an ideal vehicle for code re-use. Programming does not have to be a laborious process of writing lengthy lines of code. Rather, it can resemble the "assembly" process that has characterized modern manufacturing. Previously-defined objects can serve as components in building complex programs. A

for code re-use and is basic to its operation. However, as classes and objects are created, deleted, and modified, the image dynamically changes (see Figure 3). Such changes may have an effect on the behavior of a program.

Structural representation

In most traditional programming lan-

languages commonly used by business such as COBOL, records are of fixed length and attributes values need to be atomic. An object-oriented data model is more versatile. It supports complex data structures. The attributes of an object can take on multi-values (repeating groups) and a nested structure.

In addition, an object-oriented model differs from the traditional programming languages in the manner by which an object is identified. In the traditional programming languages, a record is referred to only in terms of its attribute values (also called key values). This identification method is "internalized" in the sense that to locate a record a program has to perform a file scan or use some attribute-based index searching scheme. In an object-oriented model, an object is not identified by its attribute values but by a system-wide unique object identifier (OID). There is no universal treatment on whether OIDs are visible to the user. In object-oriented database systems (OODB), OIDs usually are not directly visible to the user (Date, 1995). However, in some popular object-oriented programming languages such as Smalltalk, external OIDs can be assigned by and are thus known to the user (Digitalk, 1986; Goldberg and Robson, 1989). In the latter approach, an object is given an "external" name and can be referred to by that name.

Semantic expressiveness

An object-oriented model is richer in semantics than the traditional programming languages. For example, it supports generalization through class hierarchy to provide the foundation for inheritance. The traditional programming languages lack this capability. In addition, in an object-oriented model, a class is an object in its own right. It is possible to model a class as a separate entity from the individual instances of the class. A class may have its own attributes to store data and its own methods to perform actions. In the traditional programming languages, information about files, the counterpart of class, can only be represented externally. Conventional database management systems (DBMS)

rectify this shortcoming to a limited extent by storing file-related information such as record types and formats. Furthermore, files, like records, are passive entities. For the most part, they are not capable of performing actions except in some implementations where triggered procedures are allowed.

Polymorphism

An object-oriented model supports polymorphism, which allows objects to respond to the same message with their unique behavior. This feature significantly simplifies programming. In third-generation programming languages such as COBOL and FORTRAN, it is illegal to assign the same name to different procedures even though they perform the same semantic function. Programmers need to take pains to keep track of the different names assigned to procedures that are functionally identical. In object-oriented programming, the decision about which method to execute can be delayed until run-time and is based on the type of the receiver. This is called dynamic or late binding. In dynamic binding, the method to be invoked is taken care of by the receiver. Therefore, the programmer is spared of the task of writing cumbersome procedures (most likely involving case statements) for determining the type of the receiver and the corresponding method to invoke. Also, program maintenance is simplified in that the addition of new object class and method does not have to trigger changes in the other parts of the program since there is no case statement to begin with. (Interest readers are referred to LaLonde and Pugh, 1990, pp. 11-13, for a discussion on polymorphism, dynamic binding, and advantages.)

The differences between the traditional programming practice and an object-oriented model are summarized in Table 1. These differences show that object-oriented programming not only is more powerful than the traditional programming languages, but represents a new thinking about how computer systems should be built. The changes are non-trivial but incur a fundamental shift in paradigm.

Table 1
Differences between object-oriented technology and traditional programming practices

	<u>Traditional</u>	<u>Object-oriented</u>
Encapsulation of behavior	Not supported	Supported
Programming environment	Clear and narrow boundary	All previously defined class and objects
Structure	Atomic	Complex and arbitrary
Identifier	Attribute value(s)	Unique external identifier
Generalization	Not supported	Supported
Inheritance	Not supported	Supported
Polymorphism	Not supported	Supported

The next section reviews related literature. The potential internal control risks and opportunities engendered by the deployment of object-oriented technology are analyzed in Section III.

3. Literature review

Research in object-oriented accounting systems is in its embryonic stage. Very few studies have been devoted to issues in this area. Chu (1992a, 1992b) and Murthy and Wiggins (1993) studied the feasibility to apply object-oriented concepts to the design of financial accounting systems. Kandelin and Lin (1992) used object-oriented programming to develop a computational model for a managerial accounting problem.

While research in object-oriented accounting systems has barely begun, a number of models have been developed in the computer science literature for enhancing security in an object-oriented environment.

Minsky and Rozenshtein (1987) proposed a law-based approach. Prolog rules were used to specify the law of the system that governs the exchange of messages in the system. The law serves as a screen through which messages must pass.

Keefe, Tsai, and Thuraisingham (1989) developed SODA (Security Object-Oriented Database) model. The model assigns a sensitivity level to a protected object. Each message carries with it the clearance level of its sender and the current classification level of the information contained in it. Rules are developed to adjust classification levels and to determine if access should be granted to an object. In particular, *-property (Bell and LaPadula, 1976) is enforced, which requires that no object can write information it has access to where another object without proper authorization can then read it.

Hailpern and Ossher (1990) introduced the concept of multiple interfaces for objects as a mechanism to exercise access control. The idea is to provide restricted subsets of an object's

methods to specific users or kinds of users rather than make all methods universally available.

Mizuno and Oldehoeft (1990) developed an extended access control language with four-tuple entries that implements the principle of least privilege. The goal is to prevent the invocation of methods from unauthorized intermediaries.

Boshoff and von Solms (1989) developed the Path Context Model, in which the concept of "baggage" is advanced. A baggage contains "the minimum amount of information that has to be collected and must accompany the access request on its route in order that responsibility and access authority checking can be performed even though various transformations or domain crossings may occur" (Olivier and von Solms, 1992, p. 267). The baggage is verified against the recipient's security profile before any method is executed to thwart access through illegal maneuvering. In essence, baggage fulfills the same function as rules in SODA in enforcing *-property.

For the most part, these models are concerned with software techniques for protecting data in an object-oriented database system. They are pertinent to internal control. Some of these methods should be incorporated into an overall internal control framework. However, this focus on data and software techniques is a narrow one. There are other entities besides data that need protection. For example, methods need to be protected too. In addition, internal control should involve not only software techniques but also organizational and managerial policies and practices. The latter is as important as, if not more important than, the former. Effective internal control methods integrates the two.

4. Risks and opportunities

The introduction of new computer technology into organizations usually carries with it new control risks and opportunities. The new risks may stem from the nature of the technology. More importantly, they may stem from a

lack of understanding about the technology on the part of auditors responsible for designing and implementing control systems. The deployment of object-oriented technology is particularly risky because of its radical changes in paradigm from previous technologies. Most auditors are not yet familiar with it. On the other hand, the power of the new technology can be utilized to improve the implementation of internal control measures, which should also be understood.

This section analyzes the potential internal control risks and opportunities brought about by object-oriented technology. There are variations among object-oriented programming languages in their implementation of object-oriented concepts. To establish a common base of discourse, the following discussion is largely based on Smalltalk, which is considered as a "pure" object-oriented language. This is not meant to endorse Smalltalk as the best object-oriented programming language. Each language has its strengths and weaknesses. See Rumbaugh et al. (1991, pp. 296-333) for a comparison of different object-oriented programming languages.

Risks

The risks addressed here relate to potential weaknesses that an adversary may take advantage of. The internal control risks engendered by object-oriented technology stem from several sources: (1) enlarged control span, (2) structure of objects, (3) object creation, (4) object access and persistence, (5) method control, and (6) potential break-down of audit trails.

Risk from enlarged control span

In the traditional programming practice, processing power is centralized in the application program. The boundary of a program is rather clearly defined. The domain of control from an internal auditor's point of view is relatively narrow.

In object-oriented technology, processing power is decentralized and scattered in individual objects. A key feature of an object-oriented

model is to support code re-use by allowing a system to make use of previously defined classes and objects as building blocks. Therefore, an object-oriented system can access a large number of objects. To begin with, it can make use of the large number of objects provided by an object-oriented programming language. For example, Smalltalk/V (Digitalk Inc., 1986) makes available to a program over 100 pre-defined classes with over 2,000 methods. Added to this large pool are user-created classes and objects. This number of objects is substantially greater than the number of built-in procedures of a traditional programming language. The latest version of FORTRAN (Metcalfe and Reid, 1990), for example, has about 100 built-in procedures.

Therefore, the control domain for an internal auditor in an object-oriented environment can be very large. To put it differently, the number of objects that need to be constantly "watched" is large. An object-oriented program consists of a sequence of messages sent to objects. The intended behavior of the program depends not only on the sequencing of messages but on objects' responses. Irregularities will arise when individual objects that serve as building blocks are tampered with¹.

In addition to the size problem, easy code accessibility may generate another potential problem. Some object-oriented systems (e.g., Smalltalk) provide the source code of system-provided objects which is also written in Smalltalk and can be readily viewed and easily modified through the object browser. This openness is intended to improve systems development productivity by making the system more malleable. However, it may pose potential problems in control. By comparison, it would be very difficult to modify a built-in compiled procedure in FORTRAN.

This risk pertains to system maintenance rather than the initial creation of systems. The issue is how to prevent an initially sound system from being changed through illegal and perhaps subtle modification of the underlying objects that the system depends on. There are a large num-

ber of these objects whose code may be subject to easy modification.

Risk from the structure of objects

Objects encapsulate both data and procedures, which include methods to change their data and interact with other objects. Functionally, this means that access to objects results in access to both their data and the programming capability that can be used to modify their data. This condition may present a higher degree of risk than that found in the traditional programming environment not supporting encapsulation. Without encapsulation, access to programming capability is separate from access to data. Separate protections for programs and data can be set up. An adversary may need to circumvent more than one control to commit frauds.

Risk from object creation

An instance of a class is created by sending the class a message. In the case of Smalltalk, this message takes the following form: "AClass new", where *AClass* represents a particular class and *new* a method that creates a new instance of a class. Thus, the creation of objects is easy. Objects can even be created at runtime. Without proper control, this will create problems.

Risk from access and persistence

The vulnerability of objects is further increased by the ease with which an object can be accessed and its longevity in the primary storage area. An object is identified by object identifier (OID). In Smalltalk, there are two types of OIDs. One is system assigned, which is used by the system to keep track of the memory address of the object. This type of OIDs is internal to the system and is unknown to and can not be manipulated by users. The other is user-assigned to identify global objects² (Digitalk, 1986, p. 37). These are external OIDs that are known to and can be used by users to communicate with objects created in this fashion. Unless due precautions are taken, a knowledge of their OIDs

provides an unconditional access to these objects (Cox, 1986).

Global objects in Smalltalk have no exact counterpart in the traditional programming languages. They are objects whose life or scope can extend beyond a programming run. When global objects are created, they become part of an image, which encompasses all elements of the Smalltalk environment including all classes and objects previously defined or created either by the system or the user. When the image is saved, global objects are saved along with the rest of the image. They are reinstated at the next programming run when the image is re-loaded. Since global objects are part of the image, it is possible for a global object to persist indefinitely. By comparison, global variables in the traditional programming languages persist for the life of the program.

Thus, an object-oriented programming language such as Smalltalk has the capability to store objects just as database systems can store records. It is important for auditors to realize this unconventional capability. If global objects are not properly controlled, internal control risks may arise. While the techniques for controlling global objects need not be complex once the risk is understood, what is crucial to auditors is the knowledge that this type of risk exists.

Risk from method control

Polymorphism and inheritance present another source of risk. Inheritance allows objects to inherit methods from their nearest super-classes. Polymorphism gives objects a means to override inherited methods. Normally, if a method has been specified for an object, that method will be executed. If not, an available method from its nearest super-class will be executed. This normal sequence can be modified (Goldberg and Robson, 1989) in that an object has the option to use a method specified in its super-class instead of its own. To sort out which method is actually executed can be complex. More importantly, addition, deletion, and modification of methods that occur in the super-

classes can change the behavior of a program. Such changes may be subtle and difficult to detect.

Risk from break-down of audit trails

In an object-oriented environment, a direct mapping between requesters of services and providers of services is no longer possible (Olivier and von Solms, 1992). The requesters of services can conceal their identities by making requests through agent objects that act on the behalf of and may be designed to hide the identities of the original requesters of services. If this condition is allowed to prevail, audit trails would fail to capture the true sequence of events.

Opportunities

Being a more sophisticated technology, object-oriented programming also provides features that can be advantageously employed to minimize risk and enhance control.

First of all, data are encapsulated within objects. This means that requests for data have to pass through objects. Since objects are capable of executing methods, appropriate control measures can be programmed into and implemented by objects to safeguard data. Potentially, objects can serve as safe depository of data.

Secondly, in an object-oriented system, all entities, whether they are classes or instances of classes, are objects and, therefore, have the capability of participating in internal control functions. This would allow the granularity and locus of control to be flexibly adjusted to meet the need. For example, the instances of a class can exercise control over access to itself, its methods, and its attributes. A class can exercise higher level control such as access to the file where its instances reside.

Although the large number of objects in a system may increase internal control risk, this same feature may simplify control from the perspective of program verification. Specifically, the structure of each object is small in scale. Its

methods are concise as compared with the lengthy lines of codes found in programs written in a traditional programming language. Encapsulation also encourages good modularization. As a result, verifying the logic of an object's methods is relatively straightforward. In addition, inheritance provides a basis for sharing, which minimizes the re-writing of code and, consequently, the verification effort.

In addition, object-oriented technology can be used to implement powerful auditing tools. For example, among EDP auditing tools (see Cash et al., 1977; Amer et al., 1987), parallel simulation is considered as the best method for detecting fraudulent code (Cash et al., 1977). In parallel simulation, the auditor writes a program that simulate processing functions in parallel with the audited program. A constraint in using parallel simulation is the cost and the difficulty in constructing the simulation program. Object-oriented technology, with its support for code re-use, will allow the simulation program to be constructed rather quickly and inexpensively.

Furthermore, its capability to represent complex structure with ease can simplify the task of constructing structures for storing information for auditing and internal control purposes such as audit trails. As shown by Weber (1982), the construction and representation of automated audit trails are quite complex. The improved semantic capability of object-oriented technology promises to simplify this task.

The internal control risks and opportunities created by object-oriented technology are summarized in Table 2. In the next section, preventive control methods are developed.

5. Internal control methods

As a consequence of the Foreign Corrupt Practices Act, corporations are required to design and maintain adequate internal control structures, and external auditors are required to evaluate the internal control structures of client firms.

Internal control can be sub-divided into preventive controls and administrative controls. AICPA (1985) defined preventive controls as means to safeguard corporate assets and check the accuracy and reliability of accounting data. Specifically, preventive controls are designed to provide reasonable assurance that (1) transactions are executed in accordance with management's general and specific authorization, (2) transactions are recorded to permit preparation of financial statements in conformity with generally accepted accounting principles and to maintain accountability for assets, (3) access to assets is permitted only in accordance with management's authorization, and (4) the recorded accountability for assets is compared with the existing assets and appropriate action is taken with respect to any differences. Administrative controls, on the other hand, are concerned with the efficiency of operation such as analyzing deviation of actual operating results from budgets.

This section develops internal control methods in object-oriented technology with a focus on preventive controls. Three control objectives are identified: (1) to protect individual objects, which represent corporate resources, liabilities, and transactions, (2) to protect the programming structure including classes, the class hierarchy, and methods, (3) to define the audit-trail requirements for independent verification of the recording of transactions.

To achieve these objectives, *specific* internal control methods are derived from four *general* control principles (Moscove et al., 1990, p. 307): (1) valuable assets should not be left unprotected, (2) there should be a separation of related functions to introduce checks and balances within an organization, (3) there should exist a capability to re-construct the recording of a transaction for analysis and evaluation, and (4) control ultimately rests with competent employees.

Some of the control methods that have been used in the existing processing environment are still effective in an object-oriented environment. It is important to re-confirm their contin-

Table 2
Internal control risks and opportunities presented by object-oriented technology

<u>Risks</u>	<u>Opportunities</u>
1. The large number of objects greatly extends the control domain.	1. Methods can be incorporated in objects to safeguard themselves.
2. Data and data manipulation methods are stored in one place. Data tampering is made easier.	2. The granularity and locus of control can be flexibly adjusted.
3. Objects can be easily created.	3. The structure of an object is small in scope. Good modularization is encouraged. Verifying the logic of an object's method is relatively easy.
4. A global object can be easily accessed if its OID is known.	4. Inheritance minimizes re-writing and re-proving of code.
5. An object persists indefinitely in the primary storage area.	5. Object-oriented technology can be used to implement auditing tools and methods.
6. Polymorphism and inheritance make it difficult to track which method is executed.	
7. There may be break-downs in audit trails.	

ued importance in the new environment. Also, the implementation of these methods may utilize and benefit from the powerful features of object-oriented technology.

Physical Protection of Objects

A foremost concern in control is the protection of objects. The importance of protecting objects can be seen from the dual roles of objects: as storage of data and as possessors of methods for manipulating the data. Since objects contain valuable data, they are the targets of fraud. Ironically, objects in making available methods to manipulate data are also the providers of instruments for committing fraud. It is clear that objects, if left unprotected, not only expose corporate resources for exploitation but also provide means for perpetuating such exploitation.

To protect objects, two approaches are presented below: (1) protection of identity and (2) protection of access.

Protection of Identity

The goal of identity protection is to prevent unauthorized individuals from knowing the existence and identities of objects and their relationships. Both macro-level and micro-level protection measures are given. On the macro-level, object identity protection is provided through view restriction. This measure is inspired by the ANSI/SPARC architecture in database (Tsichritzis and Klug, 1978; Date, 1995), which had seen application in database security controls (Gal and McCarthy, 1985).

In view restriction, views of objects are

divided into three levels: the external level, the conceptual level, and the internal level (see Figure 4). The *external* level defines different views that particular users, particular applications, and groups of users and applications may have about the programming environment. A particular external view consists of classes and instances of classes that are relevant to a particular group of users and applications. The *conceptual* level provides a global view of the structure of the entire programming environment, namely a view of all classes, instances of classes, and their relationships. Both the external level and the conceptual level are logical - they define objects as the users and the applications see them rather than describing how the objects are stored. It is in the *internal* level that the physical representation and storage of objects are defined.

In view restriction, the strategy is to restrict a user or an application to a particular external view. Other information is hidden from them. The result is that each user or application program is aware of only a small portion of the entire programming environment and knows just enough to perform their function. No one except the object administrator is given the conceptual

view. In this way, the existence of those objects that a user and an application need not know is withheld from them.

It should be noted that there is a cost element to nearly all control measures. A cost to view restriction is that it may hinder sharing, which is an important benefit of object-oriented technology. Therefore, excessive and unnecessary restriction needs to be avoided lest the advantages of object-oriented technology over the traditional systems be lost.

Micro-level protection focuses on individual objects. Since global objects can persist indefinitely, even across programming runs, they are vulnerable. One way to protect individual global objects is to protect their external OIDs, which provide a handle for gaining access to them. External OIDs should be treated as sensitive information and should be disclosed on a "need to know" basis. As an added precaution, the naming of external OIDs should be such that they can not be easily inferred from domain information.

Another way to protect global objects is

to control their scope. Normally, when a program is activated, all previously defined and created classes and global objects that collectively constitute the system image are loaded and made available to the program or user. If it can be determined in advance which global objects are needed in a given application and when they are needed, then it would be a good practice to load global objects separately from the rest of the image. Global objects should be loaded into the primary memory no sooner than they are required and removed from the primary memory as soon as they are no longer needed. In this way, their scope or persistence is controlled. The time in which they are exposed to manipulation is shortened.

View restriction can also be applied on the micro-level. This technique would allow an object to present different views of itself to other objects (Hailpern and Ossher, 1990). In other words, the existence of certain instance attributes and methods of an object can be suppressed selectively to thwart access. Thus, the identity of an object itself is not hidden but some of its contents are.

Protection of Access

The goal of access protection is to thwart persons who are knowledgeable about the technology and the applications from gaining access to objects. The methods proposed here include environment control and password protection.

In environment control, the idea is to separate development environment from production environment. The development environment refers to the computing facilities in which the development of new systems or modification of existing systems occur. The production environment refers to the computing facilities in which application programs are run with live data. These two environments should be kept separate. Analysts and programmers whose responsibility it is to develop and maintain systems should work in the development environment only. They should be denied access to the production environment. In this way, a group of

people who have the knowledge and skill to commit frauds are impeded from doing so.

Password protection is an effective security measure that involves checking an identification code supplied by a user with that stored in the system. It is not a new but a commonly-used security measure. Objects, however, add flexibility and versatility to the implementation of password protection, because objects can store information (for keeping security code) and execute methods (for checking code). As all entities are objects, the granularity of password protection can be flexibly chosen. Password protection can be established on the class level, the individual object level, or the attribute level for controlling access to all instances of a class, an individual object, or an attribute of an object, respectively. In addition, password protection can be specified for methods to limit the access of methods to authorized parties. This would prevent the creation of new objects without proper authorization. Again, cost is a factor for choosing the granularity of control. The finer the granularity, the more costly is the implementation.

An alternative scheme to protect access is assign clearance levels to objects seeking information and sensitivity levels to protected objects (Keefe et al., 1989; Olivier and von Solms, 1992). Access is permitted only when the clearance level of the object seeking information is equal to or exceeds the sensitivity level of the protected object.

One concern in object protection is how to thwart access to protected objects through intermediary agent objects. The concern arises in situations where the original requester of information that does not have the clearance for certain information manages to obtain it through another object that has the clearance. The concept of baggage has been advanced to regulate access to objects through indirect route (Olivier and von Solms, 1992). A baggage contains information about the identities and clearance levels of the original requester of information and that of any intermediaries en route. Only when all objects

listed in the baggage have the required clearance is the information released. Such a scheme would be effective against illegal maneuvering through indirect means.

Separation of Related Functions

As mentioned above, a potential risk stems from the myriad of objects existing in an object-oriented programming environment. A change in any of these objects may modify the behavior of a system. How to guard the myriad of objects against tampering is a fundamental control issue. Here, the principle of separating related functions can be applied in managing objects in two contexts: control over design and runtime control.

Design control

In control over design, the idea is to classify objects into different classes and assign the development and maintenance of different classes of objects to different groups of people. Figure 5 presents an object classification scheme, which divides objects into two main groups: program-specific objects and utility objects. Program-specific objects are further divided into domain objects, interface objects, and control objects.

Domain objects represent concepts or things in the problem domain. If the problem domain is a financial accounting system, we will likely find such domain objects as *Cash*, *Fixed Assets*, *Accounts Receivable*, etc. A logistic system, on the other hand, might have such domain objects as *Drivers*, *Trucks*, *Warehouses*, etc. Interface objects enable the user to communicate with application programs. They consist of the graphical design, the commands, prompts, and other means that enable a user to interact with an application program. Control objects contain the control mechanisms for an application program. They start the program, present interface objects to the user, regulate the sequence of message passing among domain objects, etc. In their role, they resemble the main module in a traditional procedural program.

Utility objects serve as the fundamental building blocks for all application programs. They are either system-provided or user-designed. Examples of utility objects include objects that represent and manipulate date, time, and data structure such as bag, set, indexed collection, etc.

For control purpose, the development and maintenance of utility objects versus program-dependent objects should be entrusted to separate groups. The management of utility objects should be given to system programmers and system analysts whose responsibility is to develop and enhance the programming environment. The management of program-dependent objects is the domain of application programmers and application analysts.

Utility objects in systems such as Smalltalk are large in number. As object-oriented technology increases in sophistication, their size will grow even larger. Typically, the source code for these objects is made avail-

able by object-oriented language developers and, consequently, can be changed easily. A good control practice dictates that only system programmers and analysts should be given access to their source code and have the authority to change them. Application programmers and analysts should only be given information about the semantics of these objects (i. e., their structure and methods), can access them only in compiled form to prevent unauthorized changes.

For larger organizations, the separation of functions can be further refined. These organizations will likely have a large number of applications in each of the main business functional areas such as accounting, finance, marketing, production, and logistics. It is a good strategy for each functional area to establish a library of base objects for reuse in constructing specific application programs. These libraries can be either developed in-house or purchased from external sources. Thus, there exist three levels of objects: utility objects, functional area base objects, and application-specific objects. This arrangement is depicted in Figure 6. Each

class of objects should be assigned to a different group of personnel for development and maintenance. Again, this control measure incurs a cost. There is a potential to introduce unplanned and harmful redundancy, which may be avoided if the object administrator, who has the global view, can provide integration across functional areas.

Runtime control

Runtime control is built on separating systems design and programming from computer operations. The strategy is to prevent the person who understands the programming logic and file structures from executing the program with real data and, conversely, to prevent the person who is responsible for executing the program with real data from knowing the inner working details of the program. This control technique is important in an object-oriented environment, because an object can reside in the primary storage area for a long time and is subject to manipulation by persons familiar with the technology. Ideally, those who are responsible for running

application programs should have no knowledge about object-oriented technology. Furthermore, application programs for execution should be in compiled form. Those who are responsible for running application programs should not given access to the source code.

Determining Audit Trail Requirements

Audit trails provide the basis for reconstructing events, which can serve to deter and detect frauds. Maintaining audit trails is as important in an object-oriented environment as in the traditional environ-

Figure 6
Object classification scheme for larger organizations

ment. The issue is over the contents of audit trails. This section identifies the audit trail requirements for an object-oriented system. The implementation of these requirements depends on software.

In the traditional programming environment, the contents of an audit trail capture the interaction among the *user*, the *program*, and the *data* (Bjork, 1975): (1) The user - who run which programs, accessed what data, when, how, etc; (2) The program - program versions, paths of execution, etc; and (3) The data - before and after image of data.

In the new object-oriented environment, an audit trail has a different set of elements: the *user*, the *control object*, the *domain object*, and the *environment*: (1) The user - who accessed which objects, when, how, etc; (2) The control object - versions, execution paths, etc; (3) The domain object - methods executed, before and after image of data, etc; and (4) The environment - what objects existed in the processing environment at a given time.

There are two important differences in audit trail requirements between the traditional processing environment and an object-oriented environment. First of all, the trail itself is more complex for an object-oriented system. Figure 7 depicts the paths of processing that need to be captured for the two different environments. In the traditional processing environment, programming power is usually centralized in one main program, which may or may not call on other programs in processing data. For the most part, processing is uni-directional. Audit trails mainly focus on the before and after images of data and operations that change them.

In object-oriented technology, programming power is decentralized and fragmented. It resides not only in control objects but more so in domain objects. A user may enlist the service of a domain object directly or through a control object. One control object may call other control

Figure 7
Audit Trail Requirement

objects. In addition, a domain object may call either other domain objects or a control object. The processing path is multi-directional and recursive. Audit trails need to be structured differently from the traditional practice and should be able to answer the following questions: (1) Who initiated the request for service and when? (2) Through whom was the request relayed?, and (3) Who provided what portion of the service sought and in what sequence?

Another important difference is that, in object-oriented programming, it is necessary to keep track of the processing environment, the context in which processing takes place. There are two reasons for it. First, the structures and methods of objects may change independently of other objects. A program that requests their services may receive different responses if changes have occurred in them. Second, with inheritance, addition, deletion, and modification of a class that occurs in one or more of an object's super-classes may significantly change the object's behavior and structure. To understand the processing done by an object requires a

knowledge about the class hierarchy to which it is attached. These characteristics make it imperative that an audit trail keep images of all objects participating in the execution of a program. Each image ascertains the state of an object and the particularities of their methods including any inherited methods. Collectively, the images constitute a system picture that defines the processing environment at the time a program is executed.

Finally, there is the problem that an audit trail may be broken down, if the requesters of services can conceal their identities by making their request through agents which can hide the identities of the original requesters.

Prior research suggests ideas that can be applied in constructing audit trails for an object-oriented system. The concept of baggage is particularly useful (Olivier and von Solms, 1992). As it was originally proposed, the concept was not meant for constructing audit trails but for regulating access to objects. However, it is clear that the information contained in the baggage can serve as the basis for constructing audit trails. When a policy is adopted that prohibits modification to baggage and, thereby, makes it impossible to hide the identity of the original requester, a faithful re-construction of events can be made and unauthorized access through surrogate agent objects can be thwarted.

The complexity of audit trails in an object-oriented system requires a capability to represent complex data structure. For example, the access path to information can be varied and unpredictable and a baggage needs to carry a lot of information in a flexible structure. Being capable of representing complex semantics, object-oriented technology itself is a good choice for implementing audit trail applications.

Competent Employees

Fundamental to all controls are competent employees. The emergence of object-oriented technology imposes new demands on auditors' knowledge and skills. Auditors need to

have the capability to evaluate, control, and utilize object-oriented systems. Currently, most accounting students and practitioners are not familiar with object-oriented technology. This reflects the increasing gap that exists between the rapid advancement of information systems technology and accounting professionals' understanding of it. It is important that accounting curricula keep students abreast of the new development in information systems including object-oriented technology. It is also important that before an organization undertakes a large-scale development effort in object-oriented systems, it should make sure that its internal auditors are well-educated about the technology.

6. Conclusion

Object-oriented technology offers many advantages over the traditional programming languages. In particular, the modularity and reusability of objects promise to overcome the severe problems caused by the procedural nature of the traditional programming languages. It is expected that object-oriented technology will become the predominant programming paradigm in business.

When a new technology is introduced into organizations, new internal control risks usually arise. The risk may result from the novelty of the technology requiring adjustments in the control methods. The risk may also result from the fact that most auditors are not yet familiar with the technology and, therefore, have not yet formulated an effective strategy to control it. On the other hand, a new technology may also present opportunities for strengthening or simplifying the implementation of internal control measures, which should be taken advantage of.

7. Suggestions for Future Research

This paper analyzes object-oriented technology, studies the ensuing internal control risks and opportunities. Methods to contain the risks are developed. Some of these methods can be advantageously implemented with an object-

oriented approach. The analysis has been confined to logical and conceptual issues. Further research opportunities exist in developing specific methods of implementation. For example, it has been pointed out that object-oriented technology can be used to implement systems to satisfy the audit trails requirement. How to design such systems is both important and challenging. Opportunities also exist in employing the powerful object-oriented technology to develop new EDP auditing procedures or innovative means to implement some of the existing procedures. Furthermore, whether object-oriented technology can be advantageously applied to the design of computerized financial and management accounting systems need to be studied. In particular, object-oriented technology may provide a suitable vehicle for implementing activity-based accounting.

Endnotes:

1. Some traditional programming languages are evolving to include object-oriented concepts and consequently may face the same control risks. Microsoft Visual Basic, for example, has incorporated object-oriented features including pre-defined classes with a large number of methods. The classes provided, however, are largely limited to interface objects such as components found in Windows environment.
2. The term "global variables" is used in the Digitalk manual. These being objects, "global objects" is a better term. Global objects will not be removed by garbage collection operation.

References

1. Abrial, J. R., "Data Semantics," In J. W. Klimbie and K. L. Koffeman (eds.), *Data Base Management*. North-Holland Publishing Company, New York, 1974.
2. Amer, T., A. D. Bailey Jr., and P. De, "A Review of the Computer Information Systems Research Related to Accounting and Auditing," *Information Systems Journal*, 3-28, Fall 1987.
3. American Institute of Certified Public Accountant, "The Auditor's Study and Evaluation of Internal Control," in *Codification of Statements on Auditing Standards*, AU Sections 320.08, 320.26, and 320.27, American Institute of Certified Public Accountant, New York, 1985.
4. Bell, D. E. and L. J. LaPadula, "Secure computer system: unified exposition and Multics interpretation," *Rep. ESD-TR-75-306*, Mitre Corp., Bedford, MA, 1976.
5. Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula Begin*, Philadelphia: Auerbach Press, 1973.
6. Bjork, L. A. Jr., "Generalized audit trail requirements and concepts for data base applications," *IBM Systems Journal*, 3, 229-245, 1975.
7. Boshoff, W. H. and S. H. von Solms, "A path context model for addressing security in potentially non-secure environments," *Computers & Security*, 8, 417-425, 1989.
8. Bretl, R., D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and W. Williams, "The GemStone data management system," In W. Kim and F. Lochovsky (eds.), *Object-oriented Concepts, Applications, and Databases*, Addison-Wesley, Reading, MA, 1989.
9. Cash, J. I., A. D. Bailey Jr., and A. B. Winston, "A survey of techniques for auditing EDP-based accounting information systems," *Accounting Review*, 813-832, October 1977.
10. Chu, P. C., "An object-oriented approach to modeling financial accounting systems," *Accounting, Management, and Information Technologies*, 2, 1, 39-56, 1992a.
12. _____. "Developing accounting systems with an object-oriented language," *The Accounting Systems Journal*, 2, 90-106, Fall, 1992b.
13. Cox, B., *Object Oriented Programming - an Evolutionary Approach*, Addison-Wesley, Reading, MA, 1986.
14. Date, C. J., *An introduction to Database Systems*, Volume 1, Addison-Wesley, Reading, MA, 1995.
15. Digitalk, *Smalltalk/V*, Digitalk Inc., Los Angeles, CA, 1986.
16. Gal, G. and W. E. McCarthy, "Specification

- of internal accounting controls in a database environment," *Computers & Security*, 4, 23-32, 1985.
17. Goldberg, A., *Smalltalk-80*, Addison-Wesley, Reading, MA, 1984.
 18. Goldberg, A. and D. Robson, *Smalltalk-80: the Language and Its Implementation*, Addison-Wesley, Reading, MA, 1989.
 19. Hailpern B. and H. Ossher, "Extending objects to support multiple interfaces and access control," *IEEE Transactions on Software Engineering*, 16, 11, 1247-1257, 1990.
 20. Hammer, M. and D. McLeod, "Database description with SDM: a semantic database model," *ACM Transactions on Database Systems*, 351-386, September 1981.
 21. Kandelin, N. A. and T. W. Lin, "A computational model of an event-based object-oriented accounting information system for inventory management," *The Journal of Information Systems*, 47-62, Spring 1992.
 22. Keefe, T. F., W. T. Tsai, and M.B. Thuraingham, "SODA: a secure object-oriented database system," *Computer & Security*, 8, 517-533, 1989.
 23. Kim, W., "Object-oriented databases: definition and research directions," *IEEE Transactions on Knowledge and Data Engineering*, 2, 3, 327-341, 1990.
 24. Lalonde, W. R. and J. R. Pugh, *Inside Smalltalk*, Volume 1, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
 25. Metcalf, M. and J. Reid, *FORTRAN 90 Explained*, Oxford University Press, Oxford, England, 1990.
 26. Minsky, M., "A framework for representing knowledge," In P. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, N. Y., 1975.
 27. Minsky, N. H. and D. Rozenshtein, "A law-based approach to object-oriented programming systems," *Proceedings Conference on Object-oriented Programming Systems, Languages and Applications*, 482-493, 1987.
 28. Mizuno, M. and A. E. Oldehoeft, "An access control language for object-oriented systems," *Journal of Systems Software*, 13, 3-12, 1990.
 29. Moscové, S. A., M. G. Simkin, and N. A. Bagranoff, *Accounting Information Systems: Concepts and Practice for Effective Decision Making*, Fourth Edition, John Wiley & Sons, New York, N. Y., 1990.
 30. Murthy, U. D. and C. E. Wiggins, C. E. Jr., "The object-oriented paradigm: implications for future accounting information systems," Working paper, Department of Accounting, College of Business Administration and Graduate School of Business, Texas A&M University, 1993.
 31. Mylopoulos, J., P. A. Bernstein, and H. K. T. Wong, "A language facility for designing database-intensive applications," *ACM Transactions on Database Systems*, 5, 2, 185-207, 1980.
 32. Olivier, M. S. and S. H. von Solms, "Building a secure database use self-protecting objects," *Computer & Security*, 11, 259-271, 1992.
 33. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Englewood Cliffs, New Jersey: Prentice Hall, 1991.
 34. Smith, J. M. and D. C. P. Smith, "Database abstractions: aggregation," *Communications of the ACM*, 20, 6, 405-413, 1977a.
 35. _____. "Database abstractions: aggregation and generalization," *ACM Transactions on Database Systems*, 2, 2, 105-133, 1977b.
 36. Tsichritzis, D. C. and A. Klug, (eds.) "The ANSI/X3/SPARC DBMS Framework: Report of the study group on data base management systems," *Information Systems*, 3, 1978.
 37. Weber, R., "Audit trail system support in advanced computer-based accounting systems," *Accounting Review*, 311-325, April 1982.