Object-oriented database systems: an implementation plan

Timothy J. Waltz, David (Chi-Chung) Yen and Sooun Lee

Summarizes the basic tenets of an object-oriented database system, a brief history of object-oriented programming and the implications of the object-oriented approach

Introduction

For the past decade or so, relational database (RDB) management systems have been the industry standard, but that position is beginning to be challenged. The changing face of computing, from distributed networks to advanced graphical applications to telecommunications, has placed new demands on information systems (IS) managers. Corporate databases are forced to contend with an ever-changing and growing data set, often needing to store complex data types that traditional RDB systems have trouble handling. It is for these reasons that object-oriented database (ODB) management systems offer the promise of help to IS managers.

Proponents of the object-oriented approach feel that the technology offers many advantages over older, traditional models. By using objects to model the real world, system design can be done quicker and with a greater understanding. Also, the use of objects makes previously written code more accessible, which improves maintenance. But perhaps the most important benefit the object-oriented approach offers, which has not been fully realized yet, is reusability. Objects developed for one system can be ported to other systems and programs, and used in conjunction with those objects, with very little or no additional code written. This promise sounds very good to managers and programmers who must deal with a dynamically changing world.

Object-oriented technology is one of the faster growing segments of the computing industry. Object-oriented languages are becoming industry standards, primarily because of the popularity of C++. And when a standard is finally decided on for C++, it should only increase the popularity of the language, as well as smooth over some of the problems programmers have using C++ code on different compilers. And while ODB systems have not supplanted RDB systems yet, the fact that industry people are debating the merits of the two shows that it has made an impact. Indeed, many relational software vendors are already beginning to include object-oriented type capabilities to their database systems in order to meet the needs of their users[1]. And in 1992, 35 of the top 100 software vendors offered some type of ODB management system[2].

But if object-oriented systems are going to be the next standard, taking a share of the database market away from RDB systems will prove to be difficult. Ovum Ltd, a market research firm, forecast that sales of ODB systems will be more than $1 billion by 1996[3]. However, as recently as 1992, the ODB market was estimated at only $25 million[4]. There is a long way between the promise of object-oriented technology and the state of the technology today.

One hurdle ODB systems will have to overcome is human nature. Computer professionals are usually wary of new technology, at least until the inevitable bugs get worked out. With the proliferation of RDB systems and the countless dollars businesses have poured into setting up their current systems, MIS departments will be reluctant to throw their money at a new technology which has been marketed more towards niche markets than general applications[5]. It pays to be sure when dealing with database systems for production applications, because hidden costs of a mistake can be more costly than waiting. That is what makes purchasing and designing database applications such a long-term commitment[3]. It will be a while before IS departments begin converting to ODB systems in large numbers.

But many analysts feel “the question is not whether ODB systems will gain prominence but when”[3]. The benefits of object-oriented technology, when fully realized, will cause a dramatic improvement in the way IS departments are able to deal with information. Because of this, the development of an implementation framework is necessary in order to prepare for the coming paradigm shift. It will pay off in the long run to take the time now to look carefully at where the company is, where it needs to be, and how it can use object-oriented technology to get there. Since the technology is still relatively new, there is little information out at the moment on how companies can best prepare themselves to move over to an ODB system. There are a host of issues involved, such as

Industrial Management & Data Systems, Vol. 95 No. 6, 1995, pp. 8-17

training of personnel, upgrading of hardware, as well as restructuring the way data are stored. There are also issues crucial to database management, such as security, back-up and recovery, and access for users, that need to be addressed in the light of ODB systems.

This article will attempt to define an object-oriented database system, as well as its primary characteristics, benefits and disadvantages. An in-depth comparison of ODB systems and RDB systems will also be provided. Feasibilities will be addressed, as well as a discussion of some things that must be taken into consideration when setting up an ODB system.

Object-oriented programming

History

Previously, programming was done with an emphasis on procedures, or how to solve a problem. The standard way of solving a problem was to break it down into smaller and smaller steps, and then code each step, or procedure. But while this algorithm works well for some problems, its inherent faults became obvious during the early to mid-1980s as problems and data structures became more and more complex. Traditional programming is too linear to handle some of the complex scenarios that today's computing requires, as people try to access video, sound, and all the event-driven needs of a graphical user interface.

Object-oriented concepts have been around since the 1950s, and culminated in Europe in the language Simula67[6]. Since then, there have been two paths that object-oriented languages have taken. There are several languages that are designed from the ground up to be object-oriented, which include Smalltalk and Eiffel. The other path entails adding features to an already existing language to give it object-oriented capabilities. Languages such as these include C++, Object LISP, and CLASCAL[7].

It was only in the latter half of the 1980s, though, that object-oriented programming (OOP) really began to take off, and along with it the popularity of C++. C++ is the addition to the C programming language, which in turn was derived from two other languages, B and BCPL, all of which were created at Bell Laboratories. BCPL was developed in 1967 by Martin Richards, and B was developed by Ken Thompson and used in 1970 to create the first UNIX system. C, which was developed by Dennis Ritchie, was first implemented in 1972. Since the development language of the UNIX operating system, C has enjoyed widespread success, especially with the eventual development of the ANSI standard in 1989, which assured a machine-independent definition of the language[6]. The evolution of C++ began with Bjarne Stroustrup adding classes to C in 1980, and culminated in the mid-1980s as C++ became a fully fledged object-oriented language[8].

While still relatively new, C++ is quickly becoming the standard of the industry. Its popularity stems from its ability to solve complex problems easier than can be done with procedural programming. Object-oriented programming entails an entirely new way of looking at problems, one which many people feel addresses the needs of modern programming. The principal difference between the object-oriented approach and procedural programming is that procedures are associated with the data they manipulate: "You don't write a procedure to act on data; rather you write a method to let an object change itself"[9]. Since each object contains all the data it needs, and provides services for itself as well as other objects, objects of all different types can be combined in a program and they will know how to interact with one another, and can be combined with very little recoding. Modularity and ease of programming are the promises of OOP and the main reason it is gaining so much support.

One of the primary areas where object-oriented programming has already been applied is in the creation of graphical user interfaces (GUI). GUIs have become important for making computers accessible to all kinds of computer users, from beginners to more experienced hackers.

Programming requires complex event handling

The benefit of a GUI is that users know what to expect from a software package, even if they have never used it before, because it has a similar interface to other packages they have used. This similarity allows developers to create seamless systems, reduces learning curves and improves application usability.

But programming to allow users to use a mouse to point, click and drag, is very difficult. It requires complex event handling, and needs many different facets of the interface, such as windows, buttons and menu bars, to communicate with one another. By using the object-oriented approach, the elements of a computer screen can be divided into different objects, each with their own data and operations. In fact, there are quite a few software packages (Visual C++, Visual Basic) that allow developers to create complex and fully functional GUI with a limited amount of programming, because they tap into the power of OOP.

Developers began applying OOP concepts to develop database systems, with the first ones becoming available around 1986. Most of the early ODBs were stand alone systems, however, designed to run only on the platforms they were created for. It is estimated that only 400 to 500 of these were in place by 1990. It was not until the late-1980s and early-1990s that more advanced and portable ODBs became available. Products such as Ontos, Object Design, Objectivity and Versant Object Technology became available, each residing on the joint platform of C++, X Window System and UNIX workstations and utilizing a client/server architecture.

In the early 1990s, more fully functional ODBs were introduced such as Itasca, 0_2s and Zeitgeist. These ODBs offered more advanced characteristics, such as version support and most importantly the beginnings of fully functional DDL/DML capabilities[10]. The evolution of object-oriented technology and database systems is illustrated in Figure 1 and Table I.

Definition

One of the most important parts of OOP is identifying an object. A simplistic approach is to examine a problem and identify what the nouns are, and those will usually end up being the objects. A more rigorous definition is that an object is an entity that has certain information unique to itself, its state data, and certain actions that it performs, its internal and external operations. As an example, if a program was being written for a university, some of the objects involved would include faculty, students, courses, etc.

Figure 1. *Evolution of object-oriented technology*

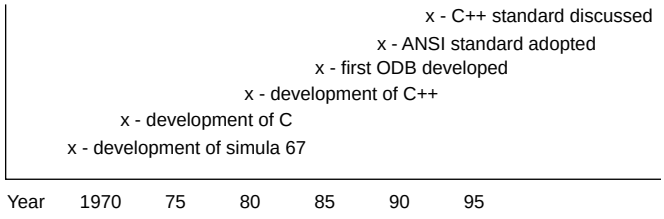
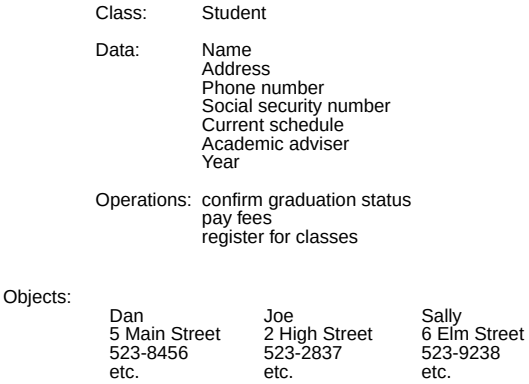


Figure 2. *Example of a class and objects*



Objects that share common operations and types of data can be grouped together into a larger category, called a class in C++ (see Figure 2). For instance, all students must register for classes, pay fees, buy books, etc. Their operations are shared by all objects – all students must do this.

In addition, all students have similar information – name, address, social security number, current schedule, past courses taken, etc. The type of information stored in each student record will be similar, even though the actual data stored there are different for each object. Classes allow programmers to treat similar objects uniformly, because they know what type of information the objects have, and what kind of services they can provide.

One of the most powerful features of OOP is inheritance, or the capability of an object to borrow or "inherit" data and operations from parent classes. A class can be created that encapsulates common data and operations of a group of objects, while still allowing those objects to have unique data and operations. For instance, if we could refine our

Table I. *Evolution of object-oriented database systems*

Year	Stage	Achievement	Obstacles
1986-1988	Beginning	Support persistent languages	Standalone systems based on proprietary languages
1989	Growing	Client/server Joint platform (C++, X Window, UNIX)	Incomplete error protection
1990-1994	Maturity	Advanced characteristics Full database features	No accepted standard Prevalence of RDB systems
The future	Prosperity	Accepted object standard Query language	Prevalence of RDB systems

student class to reflect the fact that there are different needs for different majors. The graduation requirements of an English major are quite different from those of a Systems Analysis major, and to create a function within our student class that checked the graduation requirements within each department would be needlessly complex. A much simpler answer, and the power of inheritance, is to define inherited classes, one for each department, that contain information pertinent to that department's major, such as departmental coursework, graduation requirements, involvement in departmental honour societies and so on. Each inherited class possesses data and operations from its parent class, as well as data and operations specific to itself. This allows programmers to make use of code that has already been written, and still be able to represent the unique differences between the various subclasses. Inheritance makes programming more intuitive as well as easier to do.

Another important facet of the object-oriented approach is polymorphism. Polymorphism reflects the idea that the main difference between subclasses is that they behave differently. The idea is that each object knows what type it is, for instance, whether it is an English major, a Physics major, a Psychology major, etc. Programmers using these objects do not need to know what type they are, because the objects do.

The main difference between sub-classes is that they behave differently □

Take for example the function called *confirm Graduation Status*. Let us say this function checks a student's coursework and compares it with the university guidelines for graduation. Suppose we want to check whether a student has also met his or her departmental requirements for graduation. Using polymorphism, we can also create a function called *confirm Graduation Status* for each of the subclasses as well. These functions check to see whether a student has met all the requirements for that subclass's department. The genius of this approach is that programmers using these objects do not need to know what department they are associated with – all they need to do is ask the object to *confirmGraduationStatus*. An English major would know to call the *student* function as well as the *englishStudent* function.

Polymorphism makes interacting with objects much simpler than requiring the programmer to write a long list of condition-checking clauses. By reducing code, allowing reuse of code, and making programs more intuitive, inheritance and polymorphism help reduce time spent in programming and development (see Figure 3).

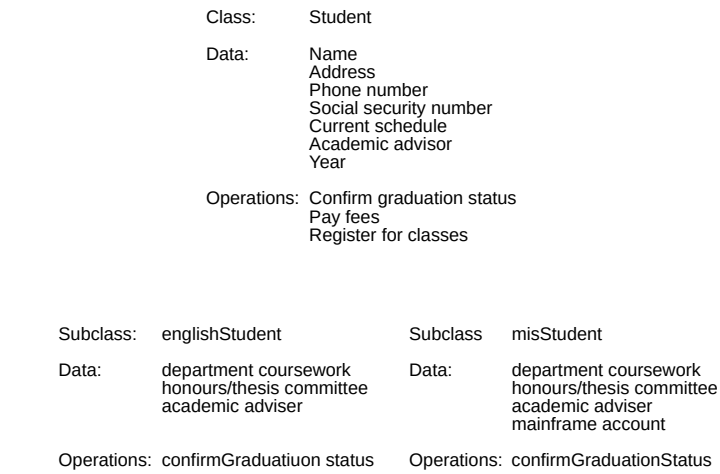
ODBs: comparison with traditional architecture

The encapsulation abilities of OOP help to overcome some of the limitations of the RDB Systems. As Thomson[11] notes, if you have a join in SQL of four keys, it requires a nesting of several different tests of keys and table calls. And if you need to add a fifth key, it could require a lot of shuffling through old code to insert the new checks and conditions.

Furthermore, the issue is not just the roundabout programming but also the inability to abstract. It is not possible, using SQL, to refer to a group of keys as an entity, even though that might make referencing data easier to code and understand. With an object-oriented approach, one could create a class that represents the groups of columns. If a fifth key is needed, it could simply be added to the class, with little need to add to the code in other classes. In this way, encapsulation makes coding more intuitive as well as easier and more compact[11].

RDBs have trouble handling complex data types, which is something that ODBs are quite good at. Edelstein[1] defines complex data structures as ones that may have "thousands of different entity types or tables, with many relationships between them, as contrasted with business applications that have relatively few tables". Since objects can be embedded in other objects, navigating through a complex structure can be as simple as accessing one object, which takes care of the rest of the navigation through internal calls and calls to other objects. This is why, until recently, ODBs have been targeted at CASE and engineering fields, where complex data structures are a way of life. Edelstein gives an example of a Defense

Figure 3. *Example of inheritance*



Department application that had 2,000 types. When run on an RDB it required 40 hours to load, while an ODB was able to load in only a few minutes. ODB advocates see this as an example of what ODBs can eventually do for all IS departments[1].

There are a variety of ways to combine the power of object-oriented programming with database systems. Since many IS managers are reluctant to have to shift their costly databases on to yet another platform, one solution calls for interfacing traditional RDBs with OOP. There are several software packages now that offer ways of doing this, such as CommonBase, Smalltalk/SQL, PowerBuilder, DBKit and Persistence.

There are several different methods for interfacing OOP with RDBs. Some of them, like CommonBase and Smalltalk/SQL, allow programmers to objectify SQL statements and joins, so that they can plug these objects into the code without cluttering up the language with SQL commands. PowerBuilder, on the other hand, is geared towards programmers who are trying to construct a Windows-based GUI. It contains Windows-based objects, which create calls to SQL underneath the builder interface. It does not offer the full object-oriented capabilities of some of the other development tools. DBKit and Persistence, on the other hand, are based on objectifying parts of the ER model. The entities and relationships are turned into objects, which in turn give access through SQL to the data[11].

Some people argue that ODBs' true potential lies not in developing as a standalone database system, but as an integration tool for bringing together a wide variety of data types. Since you can store data as well as commands in an object, it would be natural to allow users to access an object which could in turn connect them to a whole series of databases, all residing on different machines, and formatting the results to fit into a spreadsheet or word-processing program. Several ODBs already allow users to do this, or to program the capabilities into their client applications, although the technology is still in the development stage, and has failed to deliver fully on its promise[5].

Similarities and differences between ODBs and RDBs are listed in the Appendix. Ways of interfacing ODBs with RDBs are illustrated in Table II.

ODBs: pros and cons

The lack of an object-oriented industry standard, while it has not affected the optimism for the technology, has made many reluctant to consider seriously an ODB yet. Since each ODB has its own definition of what objects are and how to implement the technology, there is limited capability for objects from different systems to interact with one another. This limits the use of objects to those created by one ODB system, which is a constraint IS managers could do without[12]. Even more damaging, this counteracts the promise of modularity that object-oriented enthusiasts are emphasizing.

However, the push for a standard is growing. In fact, a standard that would allow cross-vendor portability has been proposed by the Object Database Standard: ODMG-93[13]. How soon this comes into existence will have a significant impact on the success of ODBs.

Before a standard is established, however, developers will have to deal with how objects reside in memory. Most objects created by programs are volatile – they only exist for the duration of the program or their calling function. However, in order to have an ODB, objects would need to be persistent from one running of a program to another.

As Bowman[14] says, "an ODBMS must uniquely define objects in a way that is independent of object location, properties and structure". Part of this problem is that "00 data management concepts have evolved from programming environments, rather than data management theories"[14]. As a result, ODB vendors have had to do a lot of developing in a short period of time in order to meet the needs of database users.

Table II. *Ways of interfacing ODB systems with RDB systems*

	OOP on top of RDB	Pure ODB system
Factors	Requires advanced programming techniques Will not have to change existing databases	Requires advanced programming techniques Will have to change existing databases
Advantages	Wide variety of software currently available Can connect wide range of platforms	Make full use of encapsulation, modularity Object-orientation is becoming the standard
Disadvantages	Lose some of the power of object model Tuple orientation does not fit easily with object model Cross-platform functionality not fully developed	Standard not yet developed Some applications will not benefit from an ODB Cross-platform functionality not fully developed ODBs not yet as robust as RDBs

Another road-block for object-oriented databases is that they have no ready-to-use query language. Since objects are self-contained, and must provide services for other objects that communicate with them, it is not clear how queries would be implemented. Do developers need to identify all the kinds of queries that other objects would want, and code those? Will they be able to handle unique queries that come up, or will that require additional coding? Also, inheritance creates a problem. If we query our *student* class, should we get both *englishStudent* and *sanStudent* or just *student*? Issues like this will have to be ironed out before a fully object-oriented database can be developed[14].

It is not clear how queries would be implemented

Because objects are an effective way of modelling the real world, it can really benefit areas which rely on being able to reflect a dynamically changing arena, such as engineering and enterprise modelling. In an area such as enterprise modelling, companies are discovering that, after several years of trying to build relational databases that effectively model their business, they are still not receiving benefits from that effort.

Part of the problem is that relational models are based on entities and their relationships, which are static snapshots of data and relationships that are important to the enterprise. As these change over time, it necessitates changes to the data or relationships, or creation of new entities and relationships in order to reflect the change. This can be time-consuming and, in a dynamically changing place such as the business world, it can be nearly impossible to keep up with all the changes that are taking place. The ability to plug different objects into an already existing ODB model, or to reuse code through inheritance, can help IS managers to model a dynamic process more effectively.

Also, IS managers will find it easier to keep an effective model by constructing new objects to reflect new influences and data, and plug those into their existing model in order to keep it current. ODBs can help companies have access to the information they need, even in an ever-changing environment[15].

Because the object model more effectively represents the real world in code, it also makes maintaining existing code much easier. The objects used offer keys for navigating the code and the relationships between the various objects that can dramatically reduce the amount of time it takes to understand a program.

Also, since objects are encapsulated, changes made to the internal code of one object will not affect other objects, because they are not reliant on how the object conducts itself. This would allow developers to change small parts of the code without it affecting the rest of the program. Inheritance also makes it easier to reuse code and develop new objects as needed with minimal programming time, and reduced testing time of the new system.

ODBs: a framework for a feasibility study

It is clear that object-oriented technology is going to be a major factor in the computer industry during the 1990s. Even operating systems are becoming object-oriented, as developers wrestle with how to allow users to run applications under several different environments at the same time[16].

However, while an object-oriented approach can offer such benefits as reusable code, portability, ease of programming and the ability to work easily with complex data types, there are a lot of factors which need to be taken into consideration before implementing such a system. Some of these issues include technical feasibility, software feasibility and ideological feasibility.

Technical feasibility

Owing to the different ways that ODBs store their data, most ODB systems are designed to use a client/server architecture. Whereas RDBs keep the buffer on the server, ODBs operate by installing the cache on the client. This makes traversal of the loaded database much faster than a traversal of an RDB, although loading large objects or complex objects can take a long time[1]. While most companies are moving in the direction of client/server systems already, if a company has not done this and wants to implement a fully functional ODB, there will be additional time and economic costs in getting the system ready. Most ODBs are single-server, but at least a couple of vendors offer a fully distributed version. Most of the issues involved with distributed RDBs also apply to ODBs, including query optimization, transaction management, database distribution and failure recovery, so it is possible to extend an ODB from a client/server system to a distributed one[7]. However, considerable start-up costs and delays should be expected.

There is a broad range of ODBs available for many different systems, so it should be possible to find an ODB that can be used on current computers. In fact, one of the benefits of an ODB is that a company can have a network that connects several different systems together, and the objects will know what type of system they are on and be

able to communicate with one another. While this may not be a reality yet[5], developers are working furiously to make it happen[3].

Economic feasibility

There may be considerable hardware and software costs involved in installing an ODB. Upgrades may need to be purchased to give the system more power, and more computers and communications links may be needed to hook up clients to the server. There is also the installation costs for putting the system in place, as well as the inevitable delays in getting the system running, and the cost of running the old system while the bugs are being fixed in the new one.

In addition to the hardware and software costs of moving to an ODB, there is also the cost of retraining. Since the use of object-oriented technology represents a shift in programming paradigms, there will be a need to train IS staff on the use of OOP. And as many writers note, there is a steep learning curve for OOP languages. Programmers will need to be able to develop and maintain the client/server system, as well as design the interface that clients use to access the system.

A decision must also be made about the state of the existing database system. One possibility is to keep existing databases as they are, and use an ODB to access the current databases. This is another example of the kind of thing ODBs can be very good at when an industry standard has been decided on, but at the moment it is hit or miss as to whether a vendor offers that feature fully.

Both pessimistic and optimistic locks have been proposed

Simply mapping an ODB on top of an RDB may not be the optimal long-run solution, however, because it will not offer the full reusable and modular benefits of storing data as objects. It may make sense to change the old database into a new object-oriented database, which will pay off in the long run, but will entail a potentially long and costly process at the outset.

Legal feasibility

Legal feasibility entails the ability of a company to guarantee the safety of its data. The success of an ODB in maintaining its data is linked to the ability of a client/server system to maintain security. In addition, since they put a cache on the client, an ODB must employ some type of locking system to ensure corrupted data are not written erroneously back on to a database. Both pessimistic and optimistic locks have been proposed, and it appears that a kind of hybrid of the two will be the best solution[3].

While password protection works well for small ODBs, it breaks down for large numbers of users and objects. This could be particularly troublesome to multinational companies that rely on large amounts of data flowing back and forth. Some effective methods used have been categorization schemes, which assign a group of users to a role, or controlling access to groups of objects. Hiding part of the database for certain groups of users is also an effective way of ensuring security[3].

Operational feasibility

Given that the object-oriented approach is a radically different paradigm from the traditional approach, it pays for a company to examine the type of work it does and decide whether it could stand to benefit from an object-oriented approach.

At present ODBs do not handle large-scale databases as well as RDBs do[3]. However, applications that require a series of multiway joins will probably show better performance if they are transferred to an object-oriented framework. Also, some applications that have been historically difficult to do with RDBs are proving very feasible with ODBs, including manufacturing process control, telephone switching, patient care and financial trading systems[3]. Applications users access, address their concerns, and so on until everybody who needs it has access to the system. In this way, bugs and problems can be more easily isolated and corrected while affecting as few people as possible. If possible, the old system should be left running while the new system is installed, to ensure access to data[17].

Since an object-oriented approach is a paradigm shift, it will be necessary to get people involved in the process in order to enable it to go as smoothly as possible. Enough time must be scheduled before the system is to go into place to get IS personnel trained in using object-oriented technology, and being able to develop or support the necessary requirements of a client/server system.

It is also a good idea to talk to key personnel to find out the strategic projects for the next couple of years, so that the groundwork can be laid for those early in the design phase. This will help get everyone involved and committed to the new technology[17].

What must be kept in mind when contemplating a switch to object-oriented systems is that it is an investment in the future, so it is necessary to keep a long-range view of the project, even the costs. Start-up costs could be quite substantial for getting the system running, including equipment upgrades, training of personnel, and the

expense of running the new system and the old system at the same time. But there are benefits. In one case, an IS manager felt that operating costs would be "about [one] fourth of what they were before, with better performance and integration"[17].

Also the versatility of an ODB will allow companies to react quicker to consumer forces and changes in the marketplace. Steve Jobs believes that AT&T's inability to react sooner to MCI's Friends and Family promotion was because they did not have the technology in place to handle such a program[16]. Faster and more complete performance will be reflected in the bottom line over time.

ODBs: development framework

The typical stages for software development include requirements specification, design, implementation, testing and maintenance. Most computer personnel are familiar with the basic tenets of these stages, so they will not be covered in detail, only the aspects of each one that are relevant to the development of an ODB.

Requirements specification

During this stage the reasons for installing or remodelling an existing database system should be examined, and the benefits and disadvantages of an ODB should be weighed. Questions such as what types of data structures will be accessed, how often the system will need to be changed, what types of current databases will need to be accessed, whether it is more important to be on the cutting edge or to go with an already existing technology – all these questions will need to be addressed.

Design

The single most important part of designing an object-oriented system is getting the objects right. There are several different ways of looking at problems, and an improperly designed project may not become unwieldy until it is too late to change the design easily, and there are no cheap solutions available. It pays to examine the factors involved carefully, and to decide what are going to be objects, what classes will be involved, how much inheritance there will be and of what kind, how much nesting of objects should be used, and so on: "Nothing – including object orientation – can replace good program design"[9].

The interaction of objects will also have to be carefully considered. It is important that objects be able to communicate well with one another, because that is a prime requirement of the reusability of object-oriented systems. Before coding even begins, it will be necessary to consider what types of operations an object will need to perform, what types of messages objects require from other objects and what services they need to provide. Time spent in the design of the objects will be reaped much later in the project, and will greatly aid in the development cycle.

Development

The development of the system will be tied to what type of software is being used, as well as what approach was decided on. No matter what type of database is running underneath, probably the development of the client part of the client/server system will be the development of a GUI. If an object-oriented interface is going to be used to bridge communication with an RDB, the communication will need to be set up so that the necessary databases can be accessed.

Testing

Testing really is not a stage unto itself, but more of a constant process throughout the development of the system. After each change the system should be checked to make sure it still meets with the requirements specification.

Also, the system needs to be continuously tested to make sure that it does not threaten data integrity, and that it performs as expected. This is where the object-oriented approach will come in useful, because it is much easier to develop and maintain objects because they can effectively model the real world.

Implementation

In this stage, the code for the individual objects is finally written. Each object should be tested by itself to make sure that it performs as expected, and contains all the operations that will be needed. Objects should also be checked for integrity: if objects perform a certain way in certain operations, they should perform that way in all their operations. This can sometimes be difficult to maintain if there is a complicated inheritance hierarchy.

Also, the state data of an object need to be ensured. For instance, if certain variables are set to 0 in some operations, but not set in others (even though they should be 0), then the operations should be changed so that they treat the data uniformly. There should be no surprises in the behaviour of objects, because other programmers need to be able to count on how objects perform.

In this stage, the objects should be brought together, a few at a time, and tested. It is important to take it slowly, because, if too many objects are thrown together at once and they do not work, it can often be confusing to work out where the problem is. It pays great dividends to take it slowly, working small sections at once, testing, fixing, and then continuing.

Maintenance

This stage should be helped by the use of objects, because it is easier to understand previously written code when objects are used than when a traditional procedural design has been used. Also, the use of inheritance, and the ability to plug in other objects to an existing system will make it easier to update and add features to an existing system. Also, because the internal coding of objects is not dependent on the internal coding of other objects (encapsulation), if extensive changes need to be made to one object, it will not require many changes, if any, to other objects.

Conclusions

Object-oriented technology is still new, and it is difficult to say at this time what its impact will be on the computer industry. As far as programming is concerned, it has already become an industry standard. But RDBs have proved themselves useful, and for the moment are still the database of choice for the industry. ODBs have only made inroads in certain niches of the industry. Whether that will change, only time can tell.

It is clear that the benefits offered by ODBs are here to stay. Just as every database system needed to call itself relational in order to survive in the early 1980s, every relational database vendor recognizes the potential of object technology, and the vendors are starting to incorporate some of the object-oriented technology into their databases in order to remain competitive. The ability to handle complex data structures and make use of inheritance are some of the benefits ODBs have to offer that RDBs are trying to provide as well. Where RDB vendors and ODB vendors differ is on the importance of encapsulation – it is the corner-stone of OOP, but relational database vendors may find it difficult, and unnecessary, to implement[1].

Whichever side wins, it is clear that the end result will be a hybrid of relational and object-oriented database systems, each offering a rich variety of data types, and access to a wide variety of database systems. This can only be good news for IS managers.

It is important for IS managers to examine the types of data they work with, the type of system they have, and what the needs of their company will be in the future. Object-oriented database systems offer code reusability, modularity, portability, expanded data capabilities, and the ability to interface easily with a lot of different systems. In addition, they offer all the traditional requirements of a full database system. In order to make the best use of this emerging technology, IS managers need to begin planning and laying the groundwork now so that they can take full advantage of this new technology.

References

1. Edelstein, H., "Relational vs. object-oriented", *DBMS*, November 1991, pp. 68-79.
2. Hodges, J. and Melewski, D., "Top 100: market approaches $17b; development shifting to client/server; sales up 23%", *Software Magazine*, July 1993, pp. 75-9.
3. Timo, M., "Putting object databases to work", *UNIX Review*, July 1993, pp. 73-7.
4. Varma, S., "Objects and databases: where are we now?", *Database Programming & Design*, May 1993, pp. 60-4.
5. Ricciuti, M., "Object databases find their niche", *Datamation,* 15 September 1993, pp. 56-8.
6. Deitel, H.M. and Deitel, P.J., *C How to Program*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
7. Kim, W., *Introduction to Object-oriented Databases*, MIT Press, Cambridge, MA, 1990.
8. Wang, P., *C++ with Object-oriented Programming,* PWS Publishing Company, Boston, MA, 1994.
9. Perschke, S. and Liczbanski, M., "Is OOP in your future?", *Data Based Advisor,* October 1993, pp. 49-53.
10. Bertino, E. and Martino, L., *Object-oriented Database Systems: Concepts and Architectures,* Addison-Wesley Publishers, New York, NY, 1993.
11. Thomson, D., "Interfacing objects with the relational DBMS", *Database Programming & Design*, August 1993, pp. 33-41.
12. Crane, A. and Thompson, G., "Object-oriented databases coming into their own", *Computer Shopper*, May 1993, pp. 508-9.
13. McClanahan, D., "ODBMS development", *DBMS*, November 1993, pp. 20-5.
14. Bowman, C., "Why we need object-oriented systems", *Database Programming & Design,* February 1994, pp. 27-30.
15. Due, R., "Enterprise modeling: still in pursuit", *Database Programming & Design,* November 1992, pp. 62-5.
16. Semich, J., "What's the next step after clients/server?", *Datamation,* 15 March 1994, pp. 26-34.
17. Sharp, B., "Is it OOP yet?", *HP Professional,* May 1993, pp. 20-4.

Further reading

Amaru, C., "Where object technology fits in", *Digital News & Review,* 11 October 1993, pp. 37-40.

Emigh, J., "Software forum – corporate market to reach $51b in '94", *Newsbytes*, 25 April 1994, pp. 31-2.

English, L., "The new object databases at work", *DBMS*, March 1994, pp. 66-72.

Harding, E., "ODBMSs go hybrid: HP exec explores commercial apps", *Software Magazine,* July 1993, pp. 28-9.

Higa, K., Morrison, M., Morrison, J. and Liu Sheng, O., "Object-oriented methodology for knowledge base/database coupling", *Communications of the ACM,* June 1992, pp. 99-113.

McLachlan, G., "The seven chakras of object-oriented programming", *HP Professional*, July 1993, pp. 40-4.

Premeriani, W., Blaha, M., Rumbaugh, J. and Varwig, T., "An object-oriented relational database", *Communications of the ACM*, November 1990, pp. 99-109.

Preston, A., "Object-oriented databases: the basics", *PC Week*, 28 June 1993, pp. 165-7.

Rigney, T., "Waiting for ODBMS: for object-oriented development, the future is now; but ODBMS lag behind", *Database Programming & Design,* September 1993, pp. 5-7.

Appendix: ODBs vs. RDBs – similarities and differences

Similarities

- ODBs and RDBs offer similar features such as data back-up, crash protection, security, etc.
- Both offer cross-platform compatibility – RDBs through the standard of SQL, and ODBs through the linking of objects.
- Objects correspond roughly to rows, and classes are similar to groups of rows in a table.

Differences

- ODBs are based on encapsulation. RDBs are based on referential integrity.
- ODBs are better for handling complex data types, such as sound and graphics. RDBs are better for handling large, distributed systems.
- RDBs have a standard; an ODB standard has not been fully developed yet.
- ODBs offer flexibility through the ability to combine various objects together, and to modify one object without affecting another.
- RDBs have a query language.
- RDBs are based on a mathematical model; ODBs have no similar philosophical basis.
- Methods are linked with data in ODBs.

Timothy J. Waltz is based in Oxford, Ohio. David (Chi-Chung) Yen is a Professor and Sooun Lee is Associate Professor, both at the Department of Decision Sciences at Miami University, Oxford, Ohio, USA.