5-1993

The Use of Object-Oriented Design Methodologies in Systems Design

Rick C. Larkin
Utah State University

THE USE OF OBJECT-ORIENTED DESIGN METHODOLOGIES IN SYSTEMS DESIGN

by

Rick C. Larkin

Report submitted in partial fulfillment
of the requirements for the degree

of

BACHELOR OF ARTS
WITH HONORS DESIGNATION

in

Business Information Systems

UTAH STATE UNIVERSITY
Logan, UT

1993

ACKNOWLEDGMENTS

I would like to extend a special thanks to all those who helped me through this time of my life. I would like to express my sincere gratitude to Dr. Charles Lutz of the Business Information Systems department for his continual encouragement and understanding, without which this work would not be possible. I would like to also thank my co-workers who put up with and covered for me while I wrote the final manuscript and prepared it for presentation.

Finally, my sincerest and heartfelt thanks goes to my wife, Lisa, for her patience and support in fulfilling this assignment. To her I extend a husband's gratitude.

Rick C. Larkin

TABLE OF CONTENTS

ABSTRACT

Object-oriented design methodologies, along with object-oriented programming techniques, have recently gained significant popularity as their advantages have become evident. Traditional methodologies following the structured approach have not been successful in providing an accurate model of the business system. Object-oriented techniques, where an object in the computer system corresponds to a real-world object, have been much more successful in accurately modeling the business system and its environment.

Object-oriented methodologies provide many advantages over structured approaches to systems design. Concepts such as data hiding, encapsulation, inheritance, dynamic binding, and polymorphism help to achieve an accurate and flexible system that is resistant to corruption and is easy to maintain. Objects, which are entities with a private memory and a public interface, are at the core of object-oriented methodologies. They represent the systems that are found in the real-world, and can be used, if they have been properly designed, as building blocks in various computer systems. The ability to re-use objects greatly reduces the cost of developing

new systems. Not only do objects allow the reuse of program modules, but the reuse of entire system designs as well. In addition to reuse, they promote system extensibility and flexibility.

Some drawbacks to the use of object-orientation exist, but can and should be overcome in order to obtain the advantages provided by the methodology. Students should familiarize themselves with object-oriented methodologies in order to place themselves in a better position in the job market upon graduation.

INTRODUCTION

There is a finite amount of work that can be done with traditional analysis and design methods. Even now, traditional methods are being pushed to their limits as Information Systems managers strive to cope with budget cuts, personnel shortages, and other requirements imposed by upper management. As the business world has become more complex, the ability of traditional design methodologies to model it has often broken down. The key to solving this problem is the use of object-oriented design and programming methodologies to model the business world.

Until recently, however, object-oriented design had been the domain of only a handful of programmers and designers who considered it more of a hobby than a viable design technique really useful. Recent developments have changed this view of the methodology and have brought it out into the open as a robust and preferable methodology for the design processes of both today and tomorrow. It fosters both program **and** design reusability. It additionally models the real world, within which businesses actually operate, with much greater ease than traditional methods. It offers many advantages, and relatively few disadvantages, to the business world, which makes it perfect for use from a managerial and economic point of view.

Why aren't object-oriented methodologies more widely used, then, if they provide so many advantages? The answer is simple. People resist change, and the object-oriented methodology represents change on a fundamental level. Indeed, it has been characterized not so much as a methodology as an entirely new way of looking at things. Object-oriented methodologies require analysts, designers, programmers, and anyone else involved in the systems design and development process to totally re-think the way that they look at the design process. This change is difficult for many to master, and it is the fear of this change that inhibits, to a greater degree than anything else, the use of object-oriented methodologies today.

Objectives

Object-orientation, without equivocation, represents the future of the systems analysis and design profession. Graduates entering the work force will, sooner or later, be exposed to this methodology, and those that have had prior experience with it will have an advantage over those who have not. Because object-oriented design methodologies are not yet taught at Utah State University (object-oriented programming is taught to a limited degree), it is necessary that the student acquire some knowledge of the subject on his or her own.

This study has several objectives, and aside from the learning opportunity that it has presented for the author, they are:

- Present an overview of the object-oriented paradigm, including the advantages and disadvantages that are inherent in its use.
- Compare and contrast the object-oriented methodology with traditional design methodologies.
- Review currently available design tools that implement object-oriented technology, and discuss those that were reviewed during the course of this study.
- Recommend future study that needs to be done, as well as course material that should be included in the curriculum of a graduate of Business Information Systems at Utah State University.

Procedure

Object-oriented methodologies are quite new, and there remains much to be learned about them. This study relies heavily upon technical papers and other writings of computer professionals who are actively involved in the use and development of object-oriented systems and design methods. Object-oriented tools, due to their nature, are expensive to develop and hence costly to acquire. Programs that implement object-oriented technology for certain parts of the software, but not others, are somewhat more affordable, although they are still costly. Due to the expense involved, only two object-oriented software tools were evaluated by the author: Borland Paradox for Windows® and Borland C++ with Application Frameworks. A discussion of each is included toward the end of this work.

THE OBJECT-ORIENTED PARADIGM

Until recently, systems design had been largely accomplished with the use of traditional top-down, structured design methodologies. These methodologies were originally developed to help the designer cope with the myriad of complex concepts that must be dealt with when modeling even a simple business system. Structured methodologies, along with their counterpart, the relational data model, have worked admirably well considering their limitations. Unfortunately, the business environment is not becoming any simpler, and new design methodologies are required to cope with the increasing complexity of the business world.

The System Approach

In the real world, a business entity can best be represented through the use of a systems approach or model. With such an approach, the business entity is considered to be a complete system, which can be defined as "a set or arrangement of interdependent things or components that are related, form a whole, and serve a common purpose." (Whitten, Bentley, and Barlow 1989, p 37) Each system, or business entity, can be made up of none, one, or many sub-systems, each of which is

also a self-contained system. Each sub-system can be further subdivided, and so on. If required, the concept can be expanded to include a parent company of which the business entity under consideration now becomes a subsystem, and the parent company's system becomes a supra-system to it. With this approach, as much or as little detail as is required can be accurately represented in the model.

As systems, and the business entities that they represent, become increasingly complex, traditional systems design and data modeling methodologies are becoming woefully unable to accurately represent the business entity in a viable and truly representative model. Because the business entity is not accurately modeled, the system that results is not accurate and will not provide accurate information. Additionally, traditional tools cannot even begin to represent the complex relationships that exist in many manufacturing, engineering, and graphics models, rendering them virtually useless for such applications.

Why is it that traditional structured design methodologies do not accurately model the business environment? Several reasons exist, but the principle reason why business entities are not modeled accurately is that structured design methodologies require that the internal workings of each business system be fully known and accessible within the entire computer system. In reality, businesses do not work this way. For example: employees that work in Marketing do not necessarily need to know the details of the work performed by the Accounting department employees.

They simply need to know what kinds of requests to make to acquire the information that they need. The need to know everything about every component of the system being modeled causes systems developed with traditional methodologies to rapidly grow out of control. Their increased size and complexity demands time and money to not only design them, but to maintain them. Additionally, if one of the processes ever changes, the entire system must be changed to reflect the change in that one process. Often it is easier to simply re-write the entire system rather than make a process change in the old system. These realities of structured program design lead often to high costs, overruns, and lack of reliability; three things that management seeks to avoid.

Due to the problems and high costs resulting from them, information systems personnel have turned to alternative methods of designing and developing systems to meet business needs. Object-oriented design methodologies, which until recently have only been the hobby of a few programmers and hackers, have emerged to become the most promising methodologies for business systems of today and the future. Both object-oriented design methodologies (which result in object-oriented system designs) and object-oriented data models are predicted to seriously challenge structured design methodologies and relational data models within the next five years.

Advantages of Object-Oriented Methodologies

The advantages of using object-oriented methodologies vs. traditional design methodologies can be summed up very simply: Object-oriented models represent the world in a much less abstract manner than their traditional counterparts. The only basic building block available in traditional structured methodologies is the subroutine, or sub-program, which limits the scope of such design methodologies to the description of abstract *events* (operations). Traditional methods fall short of being able to describe abstract *objects*. An object-oriented design can be constructed so as to virtually mirror the business as a conceptual system, using objects to represent individual sub-systems within the business system. This ability facilitates the construction of computer systems that model business systems and the world within which they operate.

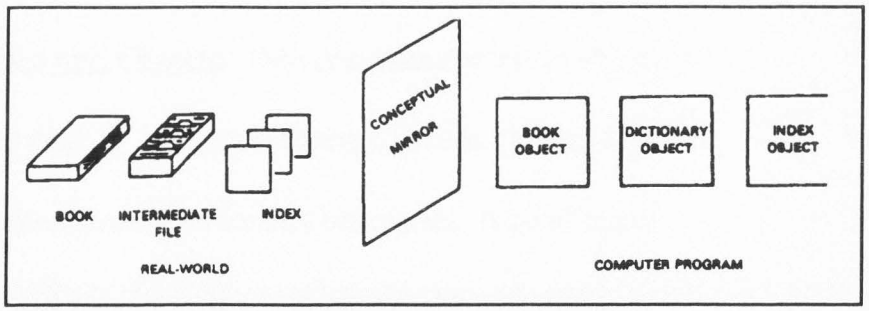

Figure 1 - Conceptual mirroring of the real world in objects. (Peterson, Ed. 1987, p 1)

Several key characteristics of object-oriented design methodologies directly contribute to this unique ability to model the business system. Some of the most important characteristics and the enhancements to systems design that they provide are listed below:

OOD Characteristic	Enhancement
• Encapsulation	- Reusability
• Polymorphism	- Extensibility
	- Maintainability
• Inheritance (single and multiple)	- Dynamic structure
• Identity references	- Dynamic data access
• Classes (and objects)	- Rapid prototyping
	- Polymorphism
	- Inheritance

Each of these characteristics of an object-oriented design methodology are important, and will be discussed in the following sections.

Classes and Objects. Two concepts central to object-oriented design are those of class and object. In general terms, a class can be defined as a set of things that have the same characteristics and behaviors. A good example of a class would be the class "car". All cars have the same basic characteristics and behaviors: they all have four wheels, an engine, transmission, etc. An object, in general terms, is simply a particular instance of a class, e.g., Rick's car. An object would contain specific

information about a particular instance of the class, such as wheel size, engine make, etc. in this particular example.

An object is an entity with a private memory (data) and a public interface (the methods that operate on that data). The data of an object can only be accessed through the object's methods, which have special privileges in accessing the data. Because objects consist of both data and the methods that manipulate the data, they provide support for heterogeneous development and tight coupling between the data and the process.

Encapsulation. The concepts of encapsulation and data hiding are natural extensions to the storage implemented by objects of both data and methods. In fact, encapsulation is what really sets an object-oriented methodology apart from other, more traditional, methodologies. Encapsulation parallels the concept of a system: that the system have a well-defined boundary and that the environment of a system communicate with it through a defined set of inputs and outputs. Objects are encapsulated in the sense that all of the data and the methods that work on the data are contained within the object. External objects cannot access the data except through the defined methods associated with the object (an object's public interface).

This particular feature of an object allows us to 'hide' data from the rest of the program. One of the biggest drawbacks of traditional design methodologies is that the data must be **global**, e.g., all parts of the program must have access to it and

know the details of its structure. This requirement severely limits our ability to implement systems that can dynamically change with the rapidly changing business environment. Object-oriented design methodologies eliminate this problem. With objects, the only methods that need know the particular structure of the data store are those that are actually a part of the object. All other objects can only access the data through the proper methods. This allows us to design dynamic and easily modified systems. When one real-world object in the business world changes, we need only modify the system object that corresponds with it. The rest of the program is unaware of and is unaffected by the change.

This 'black-box' concept of an object is central to an object-oriented methodology. Without it, an object would simply become a glorified data structure, and would be virtually indistinguishable from complex data structures found in other methodologies. It is encapsulation that makes object-oriented design and software modules reusable, which is one of the key goals of any object-oriented methodology. The system designer or programmer defines a class only once, of which an object is a particular instance. The data structure and the methods that operate on the data are also defined as part of the class. Then, any object of this class has the same data structure and same methods to work upon the data. It doesn't matter where the object appears, because it will act **exactly** the same, due to the fact that the methods that operate on the data are stored with it. As long as the proper inputs are given

to the object, the correct outputs will result, regardless of the internal process(es) used to produce the results.

This is the true beauty of an object-oriented methodology. Objects, whether they are software modules, design modules, or data stores, become building blocks which we can use in other systems. We can even change the internal control or modification processes (methods), and all objects of the same class (and all subclasses) will inherit the changes, all without the rest of the design or program being affected.

To give an example of how easily this would work, consider the notion of a table in which we wish to store information. With a traditional, structured approach, the entire program or design would be required to know how the table was implemented so that each could have access to the data stored in it. If we ever decided to change the table or its structure, we would have to modify all the other parts of the design or program that interacted with it. In some cases, this could signify re-tooling of the entire system.

Object-oriented methodologies provide a better solution. Instead of all parts of the entire design being aware of the table and all its attributes, the methods that actually work with the table, and are implementation specific, are stored with it. Examples of such methods would be sort, search, insert, delete, etc. Then, when another part of the program wished to interact with the table, a message would be

sent to the appropriate method of the table object with the required parameters. For example, if I wished to add John Doe to my mailing list, I would simply call the insert method with John Doe as a parameter. I would never need to know exactly how or where the information was stored. The methods associated with the object would take care of all such details.

As the system grows and changes, we may wish to change the table from a standard flat file representation to a B-tree representation. With structured approaches, complete re-tooling is necessary because the physical representation of the system is coded into the design; however, with an object-oriented approach, all that needs to be done is to move the data from the flat file object to the B-tree object. From that time forth, I would still send John Doe as a parameter to the insert method if I wished to add a name. Only the table object would know that the data was being stored and accessed in a different manner from the original. Object-oriented methods help to reduce maintenance costs because a complete re-tooling of a system is no longer required whenever any change, even a relatively minor one like the one described above, is implemented.

Inheritance. Virtually the only unchanging reality in the business world ***is*** change. While some forms of change may not be good, most are beneficial. Growth, in particular, is a form of change that is sought after by businesses everywhere in the continual effort to increase profit margins and market share.

Most computer systems built with traditional structured methodologies are static in nature, requiring a significant amount of time and effort to change them. In fact, the very nature of structured design methodologies tends to create designs that are rigid and inflexible. Why? Because the physical structure of the problem is woven into the program design, and ultimately into the entire system. Even if the real-world object, which the system was designed to model, changes, the computer system will often be left 'as is' due to the high cost of design changes. This results in an inaccurate model which in turn provides an inaccurate picture of the real-world system that it was designed to represent.

Object-oriented methodologies, on the other hand, lend themselves to flexibility. They tend to be dynamic in nature, easy to adapt and change. If a part of the real-world object changes, as was mentioned in the previous section, only one section of the design has to be modified to make the changes effective throughout the entire model, which in turn reduces maintenance costs. However, this is not the only type of change that is possible. It is impossible to account for all of the states in which the real-world object that we are trying to model will be found. Hence, a system needs the ability to adapt to a changing environment. Expansions to the real-world system are also possible, making this trait desirable in a system modeling the real-world as well.

Object-oriented designs cope with change extremely well. Expansions and enhancements to the existing design can be readily added, and new data structures can be built upon *already existing* structures, adding new flexibility in the way that data is viewed and handled. The key property of object-oriented methodologies that allows for change and expansion is inheritance.

As was mentioned earlier, an object is simply a particular instance of a class. All objects that pertain to a class have the same characteristics and behavior. But what happens if something changes, and you need an object that is almost, but not quite the same? With the standard approach, you must create an entirely new structure to cope with the change, and then modify the entire design to reflect this new structure. An object-oriented methodology, however, allows you to builds upon classes that have already been defined. A sub-class is derived from a class, and inherits all of the characteristics of its parent class (the class from which it was derived). The new sub-class (or child class) can then modify, remove, or add to the inherited characteristics received from its parent class. In this way, multiple classes can be created, each of which is almost the same, but differs in certain key aspects.

Consider for a moment the example used earlier of a change to a design which involves a switch from a standard flat file table to a B-tree table. With standard design tools, each data structure would have to be defined separately, and each part of the program design that interacted with these structures would have to be modified

to take into account the fact that there were two different table structures. This problem is only compounded when additional different table structures are added, such as binary search trees and hash tables. All of these tables are essentially the same, except for a few key aspects. They all store virtually the same type of data, and they all have to have the same basic set of operations available to work with it (add, delete, modify, sort, search, etc.). With an object-oriented approach, it is possible to define a single parent 'table' class and have each of the table structures inherit from it. This way, much of the structure and some of the methods need be defined only once, in the parent class. They will be passed on to the child classes as they are created. The child classes can then modify the structure and methods as needed to tailor-fit themselves to the type of tree being dealt with.

This is a simple example, but it demonstrates just how powerful a tool inheritance really is. This feature of object-oriented methodologies has the potential of saving hundreds of hours of labor on large design projects. Savings are also incurred because less modules need to be designed and less code written -- modules and code are simply inherited from other sources.

As of today, virtually all object-oriented design tools allow for single inheritance, the kind discussed in the above example. A few tools also allow for multiple inheritance. With multiple inheritance capabilities, child classes can have more than one parent class. For example, I may wish to design a car that has some

characteristics of a car and some of a truck. With single inheritance, the child class can inherit from one or the other parent class, but not both. The remaining methods and/or structure(s) must be added to the new class as a modification to the information inherited from the parent. With multiple inheritance, however, the child class can inherit from **both** parent classes, which further reduces the amount of design work and code writing required.

Dynamic Binding. For many programmers, the concepts of inheritance and dynamic binding are difficult to grasp. Object-oriented approaches are a type of bottom-up approach that seems to them as revolutionary as making the sun orbit the earth. Why? Because instead of knowing the details of a program and the structures with which it works from the outset, a programmer only works with objects in the most abstract sense possible. Dynamic binding dictates that the true nature of the object will not even be known until the last possible moment, which is often at run-time. In this fashion, an object-oriented design can incorporate a B-tree design during one run, and a flat file during the next.

For programmers that are accustomed to traditional methods, this concept is difficult to swallow. In connection with dynamic binding, inheritance dictates that an object becomes the sum total of the objects that make it up, just as a business system is really just the sum total of all the sub-systems that comprise it. The inability to know specifically what a design is before it runs is frustrating to many people.

However, the benefits that these attributes of object-oriented methodologies impart to systems design are enormous, allowing for more flexibility than was ever possible with traditional structured design methodologies.

Polymorphism. Along with inheritance and encapsulation comes polymorphism, another very important characteristic of object-oriented systems. Like the other attributes of an object-oriented methodology that have been heretofore discussed, polymorphism fits in as a part of the whole. Without the other attributes of object-orientation, polymorphism would be impossible and vice versa. With them, it contributes to the robustness of the methodology.

Polymorphism directly results from the ability of object-oriented systems to bind objects dynamically. When a program is run once, it may use a certain set of objects, but when it is run the next, it may have a completely new set of objects to work with. Polymorphism allows a design to generically call an object, and the object will behave as it is supposed to without the calling object having to do anything additional to make it work.

To use the table example again: truly polymorphic structures specify that I can send a print message to a B-tree table object, and the object itself will know how to implement the print function and will do so properly. Along the same lines, if I send the print message to a flat-file table object, it should also implement the print function properly. It should be noted that the print functions are specific to each of

the tree objects mentioned; however, the calling object need not know this -- the tree objects themselves handle the details of printing.

Polymorphism is important to an object-oriented design because it is what gives a design the modularity and maintainability needed. As was described in the section under "Encapsulation", polymorphism is the object-oriented characteristic that allows one part of a design to be interchanged with another part, all without having to modify the rest of the design that is not directly affected. It, along with the other traits of an object-oriented design methodology, provide a truly formidable arsenal in the effort to incorporate change in today's systems and to make them more truly represent the real-world objects that they are modeling.

Prototyping. Prototyping is another area of systems design that has benefitted greatly from the use of object-oriented design methodologies. Prototyping is a very powerful tool that allows analysts to work with the user in rapidly designing a system that he/she wants. Traditional structured approaches to prototyping have been difficult, at best, to implement. While some prototyping programs do exist, they tend to be complex and very expensive. Additionally, they usually cannot give an accurate representation of what the system will look like when programmed in its native environment, as the prototype of the system is running in an artificial environment created by the prototyping software.

These tools have been a great benefit to systems designers, but have not enjoyed overly wide-spread use due to the cost involved in acquiring them, and the time required to program the prototype with the software package. Now, however, object-oriented methods are helping to move prototyping into the mainstream of systems design.

A prototype requires the ability to be rapidly changed and re-run using the changed parameters. These changes are usually made during a session with the end users to get input about the system. With object-oriented technology, a prototype can be built on the platform and in the environment in which it will ultimately be run. When running the prototype, objects can be swapped easily, and due to the polymorphic nature of object-oriented designs, the program will still function properly, with only the functions dependant on the object that has been swapped or changed operating in a different manner from that originally proscribed. Program objects can be swapped, changed, and modified, all while the user is still available to provide valuable input to the design process.

The most important benefit of object-orientation to prototyping is that in most cases, the systems analyst is working with actual design objects that will eventually become a part of the working system. As objects are swapped, the system is being changed, and when the prototyping session is over, a working model of the system is in place which can be directly translated into code. (In fact, many of the objects will

already have the required code built and associated with them when the prototyping is being done.)

Drawbacks of the Object-Oriented Approach

After the proceeding discussion on the benefits of object oriented design, it would seem that there could be nothing wrong with object-oriented methodologies. Why, then, are they not being used more widely in the systems design profession? Although there are only a few reasons that could be discovered for not using object-oriented design methodologies for systems design and development, they are valid and warrant consideration when looking at the use of any object-oriented technique.

People. The reason that stands out above all others as explanation for the lack of use of object-oriented design methods is people. Humans resist change, and the use of object-oriented techniques represents a radical change from the status quo. Many analysts and systems designers have been working in the profession for several years, and see no reason for changing from 'tried-and-true' design methods to ones that have only gained popularity within the past few years.

Others see object-oriented technologies as an economic threat. One of the reasons why firms are beginning to use object-oriented technology is that it represents significant cost savings in the long run. As a library of objects is built, less and less programmers will be needed to program new objects. Additionally, less and less

system maintenance personnel will be needed to maintain current systems, as object-oriented systems are much easier to maintain and update. People, therefore, see object-oriented technology, ultimately, as a threat to their jobs.

In addition to these reasons, the learning curve for object-oriented design methodologies is rather steep. As was mentioned earlier, object-oriented methodologies require that the analyst, designer, programmer, and all others involved in the design process look at that process in an entirely different light. Not only will the program code be different, but the system design will be different. As was mentioned in the section on polymorphism and dynamic binding, the idea of delaying decisions until the last possible moment that is employed by object-oriented technology upsets those who thrive on knowing everything that there is to know about the system.

Learnability. Because object-orientation is an entirely new way of looking at things, it requires a significant amount of time to retrain personnel. There are five stages to mastering a complex object-oriented programming system:

1) Understanding the computational mechanism
2) Becoming facile with the environment
3) Learning about the reusable parts of the system that are already available.
4) Learning to decompose problems into objects.

5) Turning solutions to specific problems into generic solutions. (Peterson, Ed. 1986, pg. 186)

The first three stages are relatively easy to master, given time and patience. The fourth and fifth stages are much more difficult, with the fifth stage being the most challenging.

Understanding the computational mechanism is a relatively straightforward process. Most people have already been exposed to the ideas of objects, message passing, and inheritance. Those who have not been exposed to these concepts can grasp them relatively easy from any one of a number of texts dealing with object-oriented methods. Becoming facile in the environment is an easy process as well, requiring only that the user spend some time familiarizing him- or herself with the object-oriented environment within which they will be working.

Learning about already existing reusable parts of the system is a little bit harder than the first two steps, but once students have discovered just how much work can be saved when reusable objects are employed, they usually have enough incentive to browse around the system on their own. Additionally, an index of all the reusable objects can be maintained along with the object library(ies) to facilitate rapid search for and retrieval of desired objects.

The fourth step, it has been discovered, is much more difficult for people to understand. This is where structured and object-oriented methodologies begin to

really differ. It is one thing to understand the concepts of objects and classes, but it quite another thing to apply them. Many designers are accustomed to thinking in procedural abstractions, emphasizing actions and processes, rather than data and state. As a result, when they first try object-oriented design, they may map the process abstractions they would have created directly onto object type definitions. Others have problems with implementing behavior in the wrong objects (associating methods with a meta- (or parent) class when they should be deferred to child classes and vice versa). The underlying problem is that systems analysts and designers have had no experience with object-oriented methodologies. Presently, this skill can only be acquired through example, trial and error, or apprenticeship. There are few books that teach object-oriented methodologies on an general enough level to make them useful in this application. Methods of teaching object-oriented skills en masse to analysts are needed before object-oriented technologies can be widely accepted.

Finally, the fifth step in mastering an object-oriented design system is to turn solutions to specific problems into generic solutions. This step is the most difficult, and it is a valuable analyst indeed that can successfully apply it. It takes patience and the willingness to write and rewrite designs to make them general enough to apply to many different situations.

Technical Obstacles. The final problem that was mentioned in association with object-oriented methodologies were technical issues. Object-orientation is still a

developing methodology and standards have not yet been developed to deal with many of the situations that arise from use of the methodology, such as how objects are to relate to one another. The lack of standards also results in a lack of compatibility among object-oriented software and design tools. For example, almost any reputable relational database today can read the files from most other relational database. This is because standards have been developed to indicate how tables are to relate to one another within each database, how records within the tables relate, how fields within the records relate, etc. In the field of object-oriented databases, no such standards exist, resulting in incompatibility among all object-oriented databases that are currently on the market as each puts forth its own 'standard'.

In addition to a lack of standards, some object-oriented implementations are slow, such as multimedia caches. Multimedia is rapidly permeating the market, and many industries are looking to store multiple types of data, not just the words and numbers that can be stored in a relational database. However, graphics, sound, and other types of data require large investments in hardware to process them with any kind of reasonable speed. Advances in such multimedia storage and retrieval objects need to be made to make object-oriented technology more attractive to a wider array of businesses.

Object-oriented design methodologies are indeed the wave of the future. There are some difficulties that remain, but with patience they can be overcome.

The advantages that result from accurate business models, such as can be produced with object-oriented technology, mandates their increased use. Management is also very aware of the long-run cost and time savings that can be obtained by using object-oriented methodologies. As computer professionals, we need to prepare for the changes that will come about as a result of these new methodologies being incorporated into our work and the systems we design.

OBJECT-ORIENTED TOOLS

Tools that fully support object-oriented design are only recently being developed, with only a couple of packages being commercially produced at this time. There are many software packages, however, on the market today that *claim* to support object-oriented design or programming, but when they are analyzed more closely, it can be seen that they do not **fully** support the paradigm. Rather, they only support parts of it. The software packages that were evaluated as a part of this study fall under this latter category. All packages are excellent, and have many advantages over standard software tools. However, they do not, as of yet, fully implement the object-oriented methodology.

Paradox for Windows®

Paradox for Windows® was released in its first version in February of 1993. It is an object-oriented database product for the Windows environment running on IBM PC compatible machines. This product, as was mentioned above, was found to only partially use object-oriented technology. The product uses an object-oriented user interface, which includes all forms, reports, and other methods of viewing data.

Data can be viewed in virtually any manner desired, provided that an object can be defined within the user interface to display data in that form. The underlying database where all data is stored, however, is a standard relational database. Unlike a truly object-oriented database, objects are not stored between executions of the program; only the data is stored in a permanent form. Objects must be recreated and linked to the data each time that the program is run.

The program design of Paradox for Windows® insures compatibility with previous, non-object-oriented versions of the program, at the expense of having a true object-oriented database with the ability to store objects. The program **does** allow for storage of non-numerical and non-textual data, but does so in a file separate from the main database (using a method similar to the storage of memo fields).

Designing object-oriented user interfaces, forms, and reports was quite difficult at first. As has been mentioned earlier in this paper, the learning curve for an object-oriented methodology is rather steep. The hardest part in learning this program was in learning how objects related to one another. I discovered that it is **very** important where in the object hierarchy you place an object or a method. If, for example, you tie a method to the form object instead of one of the button objects, the method (and the accompanying action) will not be performed when the button is pushed (as was desired) but when conditions are met within the form object

(meaning that it could run when the form is opened, when it is closed, or not at all, depending upon the type of method).

Due to the graphical interface of the program, prototyping is extremely easy. Forms can be designed, changed, and tied together all while the user is present. Code segments tied to various objects can be easily modified to change the behavior of the object. This allows the user to actually see the changes, recommend additional changes, and approve the final design. I feel that this is perhaps the strongest feature of the program -- the ease with which it lends itself to prototyping.

Another nice feature of the package is that program code is generated concurrently with the design of the forms and reports that work with the database. When prototyping of the system is completed, the program itself is also completed. Unfortunately at this time, however, the finished program requires that Paradox for Windows® be present on the host machine in order to run. Hopefully, this problem will be corrected in future releases of the product, allowing for stand-alone executable programs to be created.

The graphical, non-programming type interface also restricts the user of Paradox for Windows®. Borland has supplied a number of pre-defined classes with its product. However, new classes that can be inherited by classes below it in the hierarchy are very difficult to create. New methods can be created relatively easily, but due to the lack of class creation, must be written and re-written every time that

they are needed. Additionally, the relational nature of the underlying database restricts the user to creating methods and classes that use the same basic primitive operations provided by Paradox for working with the relational database.

As with other object-oriented systems, Paradox for Windows® requires a significant amount of resources to run. For testing and development purposes, we found that an IBM compatible 486DX-33Mhz was the slowest desirable platform to run the software on. Additionally, the program will run with four megabytes of memory, but only barely. If any designing is going to take place at all, eight megabytes or more of random access memory (RAM) is recommended.

Overall, Paradox for Windows® is an excellent program. It is easy to use, has excellent prototyping capabilities, and allows code to be generated concurrently with the program design. It is difficult to create new classes to be used with the program, and these new classes must use the primitives provided with Paradox in order to access the relational database. I would definitely recommend Paradox for Windows® as a viable solution for the not-so-serious programmer that would like to design object-oriented solutions to design problems.

Borland C++ with Application Frameworks

C++ has been around for a number of years, and Borland C++ has been around for approximately four years as a newer implementation of the C++

standard. C++ is a third-generation language and is an extension of the standard ANSI C language that includes object-oriented capabilities. The Application Frameworks is a class library designed specifically for Borland C++ that includes standard classes for both the DOS and Windows environments.

Borland C++ has an excellent interface that allows the programmer to program, debug, and otherwise troubleshoot his or her C++ code. As a third-generation language, it lacks many of the graphical and other useful tools that are usually provided with fourth-generation languages. However, it makes up for this in flexibility and adaptability.

Designing object-oriented applications in Borland C++ is extremely easy when an existing class library is used. Without such a library, work in the language is tedious until a library to handle many common tasks can be built. The Borland C++ compiler generates stand-alone executable program code that can be run on any IBM compatible machine with a 286 class microprocessor or better.

Prototyping with Borland C++ is difficult, because any changes desired by the user must be made by modifying one or more methods, after which the program must be re-compiled and re-run. This process takes place with every change, which does not allow for good prototyping sessions. C++ lends itself more to a demonstration type of prototyping, where the designer asks what the user wants and then designs a system to do it. The designer then demonstrates the system, asks for any changes

to be made to it, goes and makes them, and then demonstrates the system again. This cycle continues until the user is satisfied with the resulting system.

Borland C++ is an excellent program, but lacks many of the conveniences of a fourth-generation language. However, it is flexible and generates efficient code. No database capabilities are built in, but can be created using objects. The same is true for storage of objects from execution to execution (object permanence). Borland C++ is an excellent object-oriented tool, and with an extensive class library can be very powerful. (One computer professional whom I know, using Borland C++, had a simple database application up and running within one day.) I highly recommend it to experienced users as a viable code-generation platform.

Other software products are available, including a handful of object-oriented databases. In spite of the lack of standards governing object-oriented methodologies, object-oriented databases have flourished due to the advantages that they offer businesses. Other object-oriented software, such as CASE tools, while not yet readily available, will enter the marketplace shortly as programmers and analysts gain experience working in an object-oriented environment and become familiar with the inherent advantages of the object-oriented paradigm.

CONCLUSION

Object-orientation represents the future of the computer profession. As the benefits of using an object-oriented design methodology, together with object-oriented programming techniques, become more and more evident, businesses will increasingly use object-orientation for their computer systems. This is something that as business and computer professionals, we cannot ignore. Knowledge of object-oriented methods and procedures will be vital to our continued competitiveness in the job market.

Knowledge of object-oriented methods and procedures can be acquired only through reading or attending special courses at the moment. Unfortunately, neither of these methods is a very good teacher when it comes to object-orientation, as each of them usually only impart the concept behind object-oriented methodologies, and not how to use them. The only truly effective way to learn object-oriented design methods is through guided practice.

Courses on object-oriented design methodologies are sorely needed at Utah State University in order to expose graduates to this extremely important concept before they enter the job market, making them more competitive for jobs. Initially,

I would recommend a graduate-level, **hands-on** course to introduce students to object-oriented design methodologies and the benefits that they provide. I recommend that this course be taught at the graduate-level to allow for learning the best way to teach the methodology to students on both the graduate and undergraduate levels, as well as to determine what methods would be best suited to the instruction of object-orientation.

Further research into the instructional suitability (and ultimately the learnability) of various object-oriented design tools needs to be conducted. Several packages exist on the marketplace, and occupy an entire price range from modestly priced to extremely expensive. Once again, we confront the idea that object-orientation, more than being a methodology, is a way of looking at things. I do not believe that the software package used is as important as the ability of said package to provide knowledge of object-oriented design techniques.

REFERENCES

Badgett, Tom. "OOPS: Building Applications With Objects." Personal Computing 13 (December 1989): 156.

Beaumariage, T., and Mize, J.H. "Object-Oriented Modeling: Concepts and Ongoing Research." Paper presented at the Society for Computer Simulation Multiconference on Object-Oriented Simulation, San Diego, 1990.

Booch, Grady. "Object-Oriented Development." In Tutorial: Object-Oriented Computing, edited by Gerald E. Peterson, 5-15, vol. 2. Washington: Computer Society Press of the IEEE, 1987.

Dawson, Joseph. "A Family of Models: Can Object-Oriented Databases be as Successful as Relational Databases?" Byte 14 (September 1989): 277-286.

Delcambre, Lois M. L., Steve P. Landry, Lissa Pollacia, and Jint Waramahaputi. "Specifying Object Flow in an Object-Oriented Database for Simulation." Paper presented at the Society for Computer Simulation Multiconference on Object-Oriented Simulation, San Diego, 1990.

Doyle, Robert J. "Object-Oriented Simulation Programming". Paper presented at the Society for Computer Simulation Multiconference on Object-Oriented Simulation, San Diego, 1990.

Ege, Raimund K., and John C. Comfort. "Object-Oriented Data Base Support for Simulation Environments." Paper presented at the Society for Computer Simulation Multiconference on Object-Oriented Simulation, San Diego, 1990.

Garber, Joseph R. "Here Comes OODBMS." Forbes 150 (July 6, 1992): 102.

Herring, Charles. "ModSim: A New Object-Oriented Simulation Language." Paper presented at the Society for Computer Simulation Multiconference on Object-Oriented Simulation, San Diego, 1990.

Martin, James, and Carma McClure. Structured Techniques: The Basis for CASE, rev. ed. Englewood Cliffs, NJ: Prentice Hall, 1988.

McClure, Carma. "On Object-Oriented Technology." Byte 15 (September 1990): 358.

Meyer, Bertrand. "Reusability: The Case for Object-Oriented Design." In Tutorial: Object-Oriented Computing, edited by Gerald E. Peterson, 37-51, vol. 2. Washington: Computer Society Press of the IEEE, 1987.

O'Shea, Tim, chair; Kent Beck, Dan Halbert, and Kurt Schmucker, panelists. "The Learnability of Object-Oriented Programming Systems." In Tutorial: Object-Oriented Computing, edited by Gerald E. Peterson, 186-188, vol. 2. Washington: Computer Society Press of the IEEE, 1987.

Peterson, Robert. "Object-Oriented Database Design." In Tutorial: Object-Oriented Computing, edited by Gerald E. Peterson, 180-185, vol. 2. Washington: Computer Society Press of the IEEE, 1987.

Rasmus, Daniel W. "Object-Oriented CASE." Byte 17 (December 1992): 160.

Rasmus, Daniel W. "Relating to Objects." Byte 17 (December 1993): 161-165.

Seidewitz, Ed, and Mike Stark. "Towards a General Object-Oriented Software Development Methodology." In Tutorial: Object-Oriented Computing, edited by Gerald E. Peterson, 16-29, vol. 2. Washington: Computer Society Press of the IEEE, 1987.

Sincovec, Richard F., and Richard S. Wiener. "Modular Software Construction and Object-Oriented Design Using Ada." In Tutorial: Object-Oriented Computing, edited by Gerald E. Peterson, 30-36, vol. 2. Washington: Computer Society Press of the IEEE, 1987.

Stix, Gary. "Objective Data." Scientific American (March 1992): 108.

Tesler, Larry. "Programming Experiences." In Tutorial: Object-Oriented Computing, edited by Gerald E. Peterson, 67-71, vol. 2. Washington: Computer Society Press of the IEEE, 1987.

Trevallyn-Jones, Nik. Letter to author. 25 May 1993.

Whitten, Jeffrey L., Lonnie D. Bentley, and Victor M. Barlow. Systems Analysis and Design Methods, 2nd ed. Homewood, Ill.: Irwin, 1989.

"Look Out, SQL: Here Come Object-Oriented Databases." Byte 13 (July 1988): 11.