Object-oriented programming

by R. P. Ten Dyke
J. C. Kunz

Object-oriented programming involves a new way of thinking about and programming applications. The thought process and techniques are introduced through a discussion of the language Smalltalk and through an illustrative example. These concepts are extended to a hybrid functional language, object-oriented system, KEE®, and illustrated through the use of knowledge-based system examples.

Object-oriented programming can trace its roots to the earliest uses of the computer, yet it is thought of by many as a new programming methodology. It has been implemented as a programming language, and its ideas and constructs have been added to existing languages. Still, many who work with computers are unfamiliar with object-oriented programming and how it is distinguished from more conventional programming methods. Object-oriented programming is a methodology that employs data abstractions, called *objects*, as the basic structure of the program. A data abstraction is a way of defining data by the way in which it may be used, rather than by what it is. Goldberg and Robson[1] propose the following three views of object-oriented programming:

- A *vision*, or a way of organizing a system description. The central intuition in the object-oriented vision is that systems can be built by describing sets of related objects and that objects have attributes and behavior.
- A *set of programming techniques* that specifically includes techniques to manipulate objects, attributes, and behaviors.
- A *large complex system* that enables and encourages programmers to create applications using the object-oriented vision and that provides techniques to manipulate objects, attributes, and behaviors.

This paper describes object-oriented programming as an important and continuing evolution of programming methodology and focuses on its use in developing knowledge systems. Our purpose is not to achieve the technical depth that would be achieved through user manuals for specific object-oriented programming languages or through other means. Rather, it is our purpose to put object-oriented programming in perspective so as to give readers a clearer view of where it fits into the program development picture and an understanding of some of its important benefits.

Early background

The basic principles of object-oriented programming date from the early commercial uses of computers. For example, the Minuteman missile was designed by computer in 1957, using primitive object-oriented techniques. The design was divided into a handful of related discrete components. One component dealt with the structural design. Another considered drag for the portion of flight that took place in the atmosphere. A third component dealt specifically with nozzle design to determine thrust under various speed and atmospheric conditions. A trajectory component had two parts: powered flight (the engines firing) and free flight (all fuel exhausted). Additional components included one that would oversee the whole operation, including the detachment of fuel tanks as the fuel in each tank became exhausted. The system provided for the passing of information between the components. For example, the trajectory component might ask the nozzle component which thrust levels to use for each speed and altitude, and the structure component might ask the nozzle component what weight to add to the structure.

Object-oriented programming is like having a group of specialists working together to solve a problem.

Each of the components was created by a specialist who had a unique expertise, for no single individual had the breadth of knowledge to create the whole program. Each specialist determined what information would be needed from the others in order to complete one portion of the design. Each of the program components was encapsulated, that is, it had its private data and was virtually a separate program with its own methods and procedures that would apply to those data. By sending data and commands between the components and by using key design parameters supplied by the operator, the computer program designed and simulated the flight of experimental missiles in the computer. After a large number of these experiments, one design specification was chosen by the engineers.

Object-oriented programming as we currently conceive it was not known or used in 1957. However, the current term *object* is a suitable substitute for the term *component* of the missile design program. The missile specialists effectively developed the design program as though they were working together, passing information back and forth. Similarly, in object-oriented programming, the objects are made up of private data and methods, and they communicate with each other through the use of *messages*, which may contain *data arguments*. In this way, object-oriented programming is like having a group of specialists working together to solve a problem.

In the missile design program, it was necessary to design a system for keeping track of the status of each of the components and for determining which information to send where. In today's object-oriented programming, these functions are formalized and generalized so as to apply to many different kinds of objects.

A need to find more general solutions to simulation problems and the formalization of the concepts of data abstraction led to the development, in the 1960s, of an Algol-based language, SIMULA67.[2] This language introduced the concept of *class*, which is the implementation of a data abstraction through encapsulation, and the concept of a *class hierarchy* to permit the inheritance of methods. This generalized the approach used by the missile design program to break a complex problem into smaller, more manageable parts.

It was important to the development of object-oriented programming to recognize that data abstraction was suitable for many different classes of problems, not just for simulation. Using SIMULA67 as a springboard, the language Smalltalk was developed at the Xerox Corporation's Palo Alto Research Center. Smalltalk has further formalized the notion of objects and message passing among objects.

Other languages have been developed by adding object-oriented constructs to existing procedural or functional languages. For example, the languages EIFFEL,[3] C++®,[4] and Objective-C™[5] are extensions of the C language. FLAVORS[6] is an object-oriented extension to LISP, and LOOPS[7] and KEE®[8] are programming environments that use object-oriented programming as a functional building block. It is also possible to consider object-oriented programming as a programming style, rather than as a language, and build object-oriented programs using existing procedural or functional languages.

There is no complete agreement on all of the functions that must be present for a language or system to be called "object-oriented." In 1987 Peter Wegner proposed that the term *object-based* be used for languages supporting objects, and *object-oriented* for languages supporting objects, classes, and inheritance.[9]

An *object* consists of data that are tightly coupled with all of the operations that can act against it. The operations are often referred to as *methods*, and the communication between objects requesting that some action take place is often referred to as a *message*.

A *class* is a template from which objects are created.

Inheritance is a property of classes that allows them to share resources. Classes may be arranged in a hierarchy from most general to most specific. Classes lower in the hierarchy may inherit methods and attributes from classes above.

Other properties that are often provided by implementations of object-oriented programming are dynamic binding and encapsulation.

Dynamic binding is a capability that is not unique to object-oriented programming. However, some object-oriented implementations provide additional generality by providing late or dynamic binding, which allows storage to be defined at run time rather than at compile time.

Encapsulation is a term that describes the scope of unrestricted reference to the attributes of an object. In general, an object can examine and modify all of its own attributes. However, it may be desirable to restrict the freedom of other objects to retrieve or replace its attribute values. To provide access to attribute values, an object can provide accessor functions to allow other objects to inquire about its attribute values or to change them. Accessor functions control access to preserve privacy and integrity of attribute values by allowing them to be read and replaced only when appropriate. They can also provide traps to respond gracefully to ill-timed or ill-formed requests.

Object-oriented programming concepts may be implemented in varying degrees of completeness. These concepts have been used as the organizing principle for an application development. They have also been used to supplement existing languages.

Smalltalk

In Smalltalk, all data elements, including integers, are considered as objects that have associated methods. For example, the operation "2 + 3" is interpreted as sending the "+" message with the argument "3" to the integer object "2." The Integer class has a method "+" that is inherited by the instance object, "2," which performs the addition and returns the sum "5" to the sender.

There are also file streams, windows, and graphic images available to the application developer.

The object classes in Smalltalk are arranged in a class hierarchy. For example, the Integer class is a subclass of Number, which in turn is a subclass of Magnitude. The class Number also has a subclass Float for floating-point numbers and a class Fraction for exact representations of rational numbers. Similarly, the class Integer has subclasses for large and small integers. Each of the subclasses of Number responds to the message "+," although the exact implementation of each might depend upon the class in which it is defined. When a message is received by an object, it looks first for a method definition to the methods of that particular class of which it is an instance. If none is found, the search continues to the next higher level in the hierarchy until the appropriate method is found.

The Smalltalk class library also contains a class Collection that provides for subclasses that are collections of numbers or other objects, including Sets, Arrays, Strings, and others. There are also classes for file streams, windows, and graphic images. All of these are available to the application developer.

Everything in Smalltalk is implemented as an object. There is one super class, called Class, which is the root of the class hierarchy tree. Smalltalk itself is implemented in Smalltalk, being built upon a core

Table 1 Metes-and-bounds description of a plot

Feet		Degrees	Minutes	Seconds	
60.00	south	2	26	50	west
244.84	south	25	0	0	east
121.90	south	65	0	0	west
30.04	north	55	7	57	west
114.72	north	25	0	0	west
139.74	north	58	2	0	west
168.59	north	31	58	20	east
112.07	south	87	33	10	east

of functions that are directly implemented in machine language.

The language has tools to allow the application developer to develop his own classes. These might be subclasses of existing classes. One might develop a special-purpose array as a subclass of the class Arrays, and inherit the use of all methods that are presently defined for Arrays. Or one might develop a class at the highest level of the hierarchy as a subclass of Class, and make use of methods that have been created by the developer only.

An illustrative example

The vision of object-oriented programming is that complex systems are created as sets of related objects and that objects have attributes and behavior. In the missile example, the objects were components of a rocket system—nozzles, trajectories, and structural design. Some of these objects described physical entities, such as a nozzle, and others described abstractions, such as trajectories. We often find that systems can be represented effectively by combining both conceptual and concrete objects. Attributes of these objects include such parameters as weight, diameter, trajectory profile, and thrust profile. The objects in the Minuteman system had behavior in that they could request information of other objects—such as thrust level—to use in a particular situation. They could compute new values for attributes and store the computed values.

This background example suggests a general theme in the vision of object-oriented programming. In developing an application, the programmer asks for the following information:

- *Objects.* Identify the conceptual and the physical objects that people specify to describe the problem area.
- *Attributes.* Name the features and important properties of the object.
- *Behavior.* State the action or function performed by the object and what users or other objects can ask of the given object.

Our next illustration is an application that involves external data. The data items for this application are stored in a file and brought into the object-oriented programming system. They are converted into object instances within a class. On a plot of land we want to define a home site that conforms to the town's zoning laws. The law says that the house must be at least 50 feet from any boundary. As a surveying tool, we want to develop an application to determine the exact dimensions of that portion of the plot of land on which a house may be built.

Start by thinking about the physical objects to be dealt with, beginning with the plot of land. To describe the land as data, the example uses a legal description that has been copied from the deed. Table 1 gives a description of the actual plot in terms of its metes and bounds. A metes-and-bounds description is a way of specifying land that is based on the existence of a reference monument or marker, such as a stake in the ground. Starting at that marker, a surveyor walks around the perimeter of the property along a series of straight-line boundaries, ending each straight line with another stake or reference marker.

A surveyor uses a specialized way of describing direction, based on the four points of the compass. Imagine that at each stake the surveyor faces either due north or due south, then turns in some direction—either east or west—an amount no more than 90 degrees, and proceeds in that direction for the required distance. Degrees, minutes, and seconds are used for the measure of all angles.

Next, imagine a particular surveyor with whom we communicate only by sending messages. Let the first message be our property description and the request, "Send me a description of the legal building site on this property." The surveyor has technicians who will help on specific tasks.

Thus we construct a program in Smalltalk that includes object classes and instances that behave much the same as the technicians in the surveyor's organization.

In the program we create a class, Plots, that understands metes-and-bounds descriptions of land. The

class *Plots* knows that a plot is a list composed of straight-line boundaries. Table 2 gives a list of object *classes* in the example plot application.

We add a class, Stakes, which uses a traditional x-y coordinate system to determine the location of each stake. Thus, for example, the attribute FeetEast is the displacement in the x direction, and FeetNorth indicates the displacement in the y direction.

We create another class, Boundaries, that understands line boundaries. A Boundary has three attributes, a Stake (as just described), a Distance, and a Direction. When it converts the legal description into an instance of the class Plots, the program creates and adds the stake position data to the distance and direction given in the plot description.

Finally, a class Directions knows that a direction contains five elements (attributes): (1) either the word "north" or the word "south," (2) an integer ranging from 0 to 90 degrees, (3) an integer in the range of 0 to 60 minutes, (4) an integer in the range of 0 to 60 seconds, and (5) either the word "east" or the word "west."

Each class has a list of messages (with corresponding methods) to which it can respond, to allow for communication between objects. We also note that a class may contain attributes that are members of some other class. For example, Plots knows how to respond to the message "shrink" and responds with a metes-and-bounds description of a plot that is smaller. "Shrink" is the message that corresponds to a method in the Plots class. To solve the building-site application, the program user sends Plots the message "shrink 50 feet," because a legal building site requires 50 feet of clearance from each boundary.

The exact syntaxes for sending messages in various object-oriented languages vary, but they can resemble the assignment-statement-and-function-call syntax of traditional languages. In Smalltalk, a message to shrink the plot might read as follows:

```
Plot1 shrink:50
```

where Plot1 is an instance object of the class Plots and is called the *receiver* of the message. *Shrink* is the message, and the colon followed by 50 indicates that the number 50 is an argument to be used in the corresponding method. Implementing the method returns a new plot that becomes a new instance of the class Plots, called Plot2, as follows:

Table 2 Object classes in the example plot application

Class	Private Data
Plots	A list of Boundaries
Boundaries	Stake, Distance, Direction
Directions	North/south, degrees, minutes, seconds, east/west
Stakes	FeetNorth, FeetEast

```
Plot2 = Plot1 shrink:50 .
```

To implement the "shrink" method, Plots will need to ask Boundaries for a boundary that is turned 90 degrees and has a length of 50 feet.

Boundaries, in turn, asks Directions for the new direction with a 90-degree rotation.

Plots, Boundaries, Directions, and Stakes are all subclasses of some parent class. Each of these is an ordered list of items, so one implementation is to make each of them a subclass of the Arrays class. The Arrays class is a pre-existing class that comes with the Smalltalk language and has methods for dealing with ordered lists. By making Arrays the parent, Plots, Boundaries, Directions, and Stakes inherit all methods and messages that are understood by the class Arrays. This includes the message that requests a return of that item which is located in any particular integer position on the data list. Thus a message to return the third item to Plots causes the return of the third Boundary on the plot.

Similarly, to know the Direction indicated by a Boundary, send Boundaries a message to return the third item to obtain the correct response. However, this approach violates the encapsulation principle of object-oriented programming. A piece of data should not have its type or meaning determined by its location in a list. This principle distinguishes objects from data. At some future time, a programmer may change the internal representation of data in the Boundaries class. If, as a result, Direction became the second or fourth item, requesting the third to get that information would yield an incorrect result. Users of data would then become responsible for understanding changes that take place in internal data format. All methods using the message to obtain Direction based on its position would now have to be identified and changed, leading to a potentially expensive and time-consuming maintenance bur-

den. Rather, it should be the responsibility of the person who makes the change in the internal data format, and only that person, to protect the users from the effects of that change.

To protect the data user, we create a method to allow Boundaries to respond to a message "direction." At the same time, we can disable any message in the

Messages and methods give the classes the ability to work together.

Boundaries subclass to return data by position by creating a method for Direction that supersedes the one which would otherwise be inherited.

Now, when we send the message "direction" to Boundaries, Boundaries responds with the element that represents Direction at that moment. If the internal representation changes, only the person making the change needs to change Boundaries' interpretation of the message "direction." All classes that send the direction message to Boundaries continue to receive the correct response.

Certainly, another approach would be to define Boundary not as a subclass of Array, in which case, none of the Array methods would be inherited.

Using the word *direction* to mean both an object and a message causes no confusion, because the syntax of our statements makes it clear when a word is meant as a message and when it is meant as an object. To make the object-message distinction easier, a convention has been adopted in Smalltalk. When a symbol represents an object, start it with a capital letter (Direction); when used as a message, start it with a lower-case letter (direction).

Messages and methods give the classes the ability to work together.

Another Smalltalk class, Pen, operates like a pen on a drawing table. We can lift it, move it, drop it, and draw. Using this class, we are able to translate the symbolic representation of the plot of land into a graphics one. Figure 1A is a drawing of the original plot of land. Figure 1B shows each of the boundaries moved toward the center by 50 feet.

The eye immediately sees the possible building site enclosed by the new parallel lines, but some additional calculation is needed to get a complete description. This is shown in Figure 1C by using a method we created to resize each boundary to the length that is defined by its intersection with the next boundary. The eye immediately goes to the southwest corner of the new plot, because there seems to be something wrong with it, and indeed there is. We failed to recognize that the new plot takes on a different and simpler shape than the original. Using an editing method, we delete the offending boundary, recalculate the intersection, and the result is shown in Figure 1D.

By treating graphics as objects, object-oriented programming makes mapping symbolic information to pictorial information relatively easy. It is well understood that charts, diagrams, and icons can enhance the understanding of data, and this example again demonstrates that presentation can dramatically affect how quickly data can be perceived.

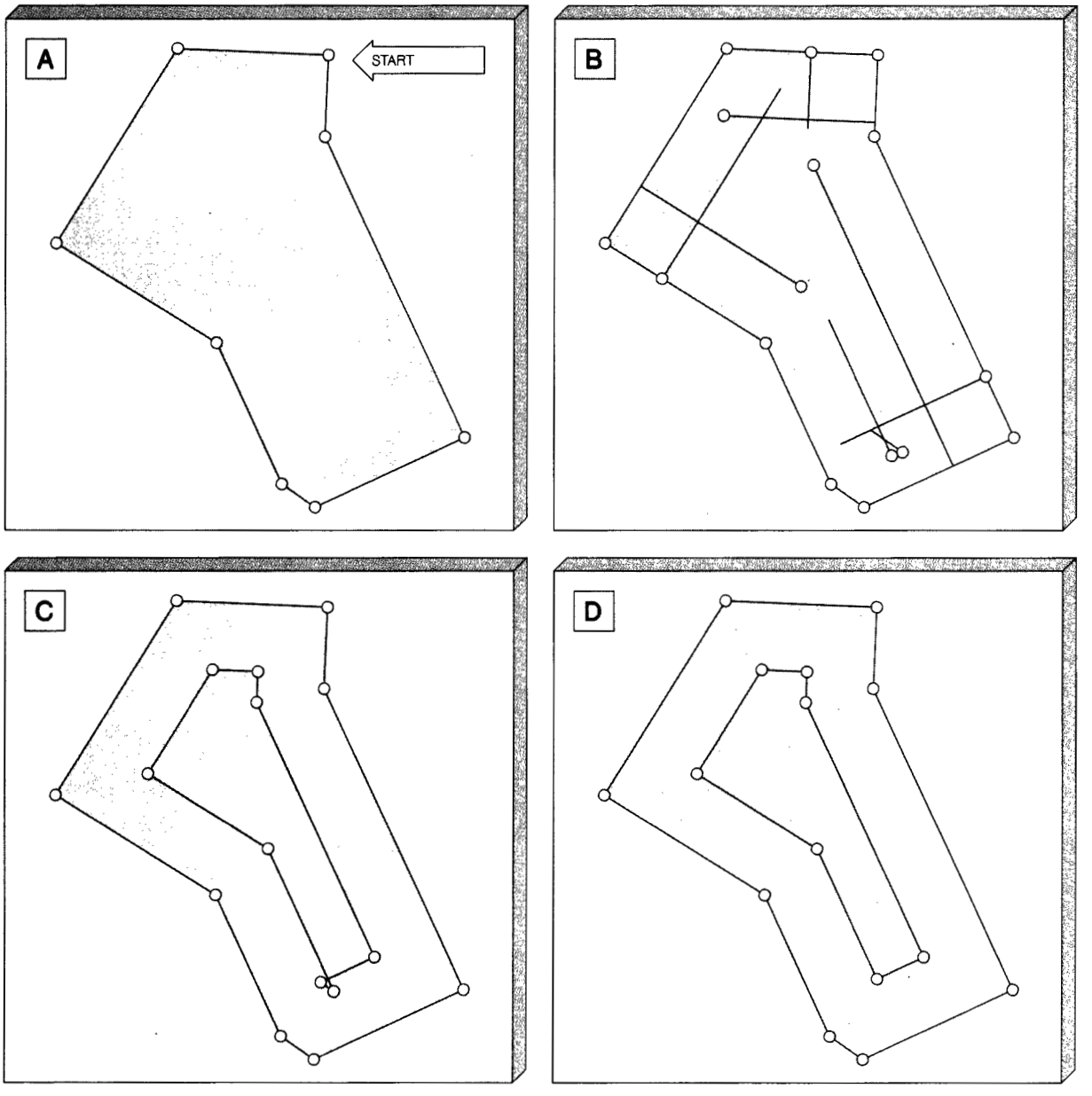
This simple example has shown how object-oriented programming might be used to create an application program. If, however, object-oriented programming were to be used to develop many applications of a particular kind—knowledge-based systems, for example—one would expect that special tools could be developed and made a part of the class library. Knowledge-based systems are heavy users of rule-based inferencing and frames, so it would be reasonable to add these capabilities to the basic language.

Knowledge Engineering Environment

The development of the tool Knowledge Engineering Environment™ (KEE) was a response to this need. KEE is a development environment for knowledge systems that uses object-oriented programming. Developed by IntelliCorp, Inc., the intended user of KEE is the developer of knowledge systems at the high-end of the complexity scale, which includes applications of planning, scheduling, and product or systems design.

In one application, for example, an airline has determined that it can improve its revenue and profit by offering a certain number of reduced-fare, no-refund seats for its flights. Each flight has a certain number

Figure 1 (A) Plot drawing; (B) plot drawing with parallels; (C) building site as newly described; (D) revised building site



of these reduced-fare seats allocated. Experts in fare structures assign the number of each fare category of each flight, depending upon the time of day, day of week, competition, nature of the market, and the airline's business goals. It is a function of airline fare policy to determine, on a flight-by-flight and day-by-day basis, the number of these special seats to allocate. More seats are allocated on flights that are not expected to have a heavy demand for the regularly priced seats. Further, these expectations may change as the date and time for the flight approach.

Historically, this allocation policy has been carried out manually. Using KEE, a knowledge-system user

designed and automated parts of this policy. The implementation of the policy will remain unchanged. Thousands of reservations agents will use a transaction-oriented computer system to determine the price a customer must pay for a particular reservation, but the count of available seats in each fare class will be updated by a knowledge system rather than by a human expert. The application runs on a LISP machine with links to an operational transaction processing system.

Brief descriptions of some other applications of KEE are given in the box on the right.

The principal goal of the KEE system has been to offer high performance and high function within a single environment. KEE is a *hybrid system*, that is, one that permits the combination of a number of problem-solving techniques. These include the use of frames, rules, a powerful programming language (LISP), graphics explanation facilities, and data-directed control, because no one technique is sufficient to support the goal of high-performance and functionality. Later versions of the product add a multiple-worlds facility to maintain separate contexts. The advantage of using the object-oriented approach has been to allow evaluation of the system to occur in an orderly manner.

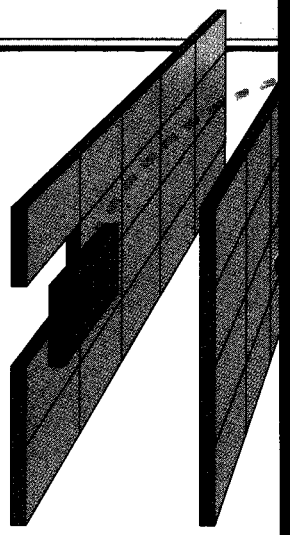
In the sense that Smalltalk is written in Smalltalk, KEE is written in KEE. A core is written in LISP, but much of the system's capability is provided by KEE knowledge bases. For example, the interface includes objects that display and manipulate objects, and the inheritance capability is a set of objects that have methods and LISP functions.

As a design philosophy, KEE employs object-oriented programming in the same ways as Smalltalk. Each of the major methodologies has been implemented using object-oriented architecture. As a programming facility, the capabilities of object-oriented programming used in the system implementation are also passed through to the user.

The basic object within the KEE environment is called the *Unit*, which is roughly comparable to the concept of a class in Smalltalk. Like Smalltalk, KEE Units are organized in a hierarchy with inheritance properties. However, unlike Smalltalk, a particular KEE Unit may be thought of as being an element in more than one tree hierarchy simultaneously. Since a KEE Unit can inherit methods and values from KEE Units at higher levels, the KEE structure allows for multiple inheritance paths.

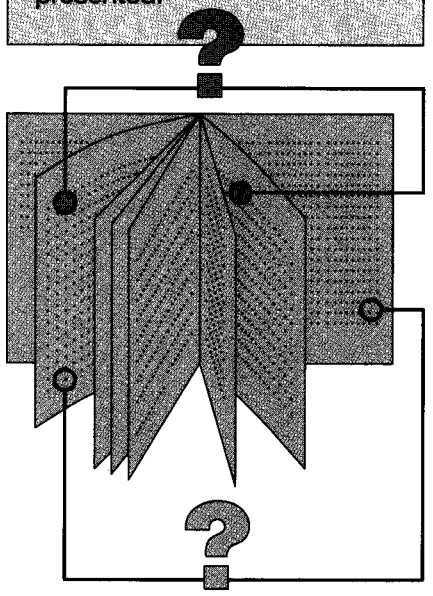
Representative KEE Applications

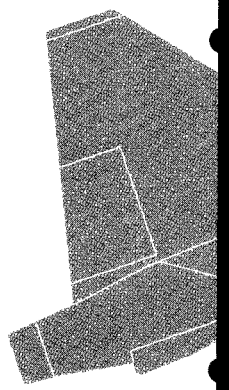
The KEE system has been used for many kinds of knowledge-processing applications. Three representative examples are the following diagnostic, analysis, and planning applications:

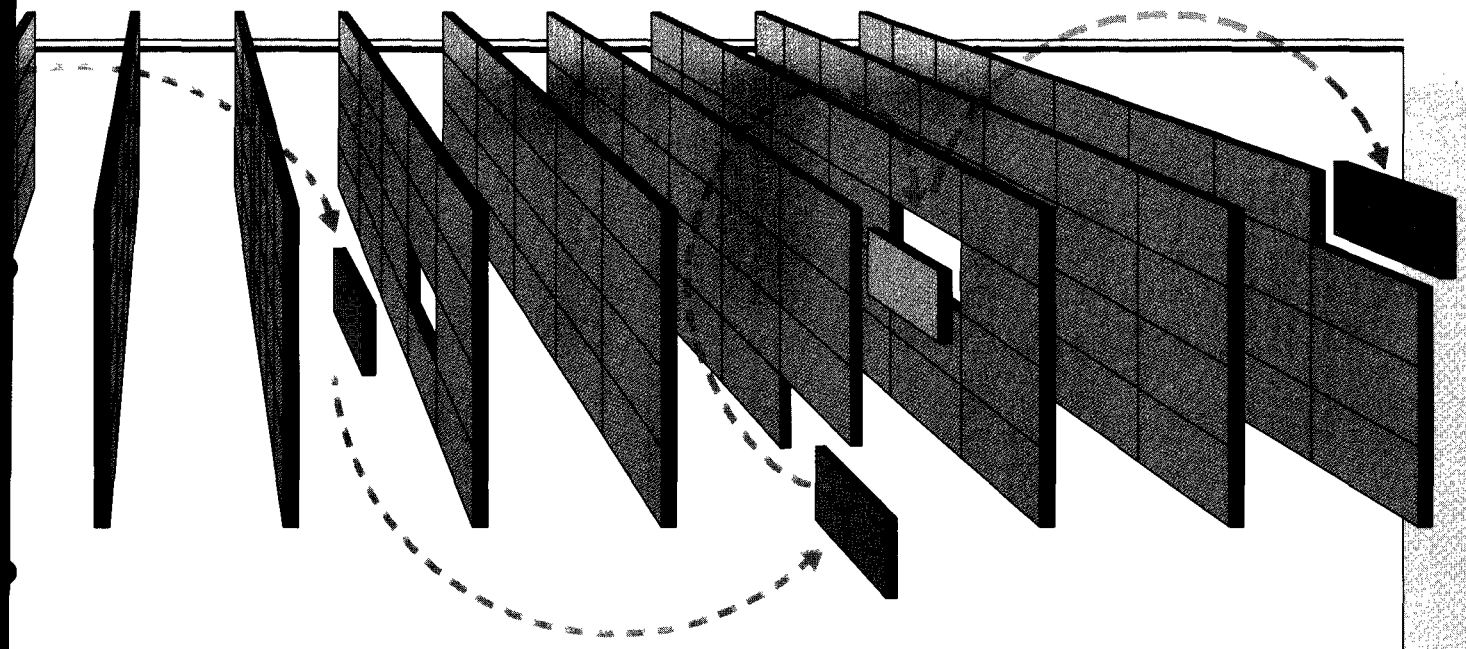


Financial system analyzer

Arthur Andersen & Co. developed the Financial Statement Analyzer. The system enhances productivity of human analysts by reviewing financial statements including parenthetical remarks, footnotes, and the management discussion and analysis. The system performs consistent financial analysis despite variations in the way data are presented.

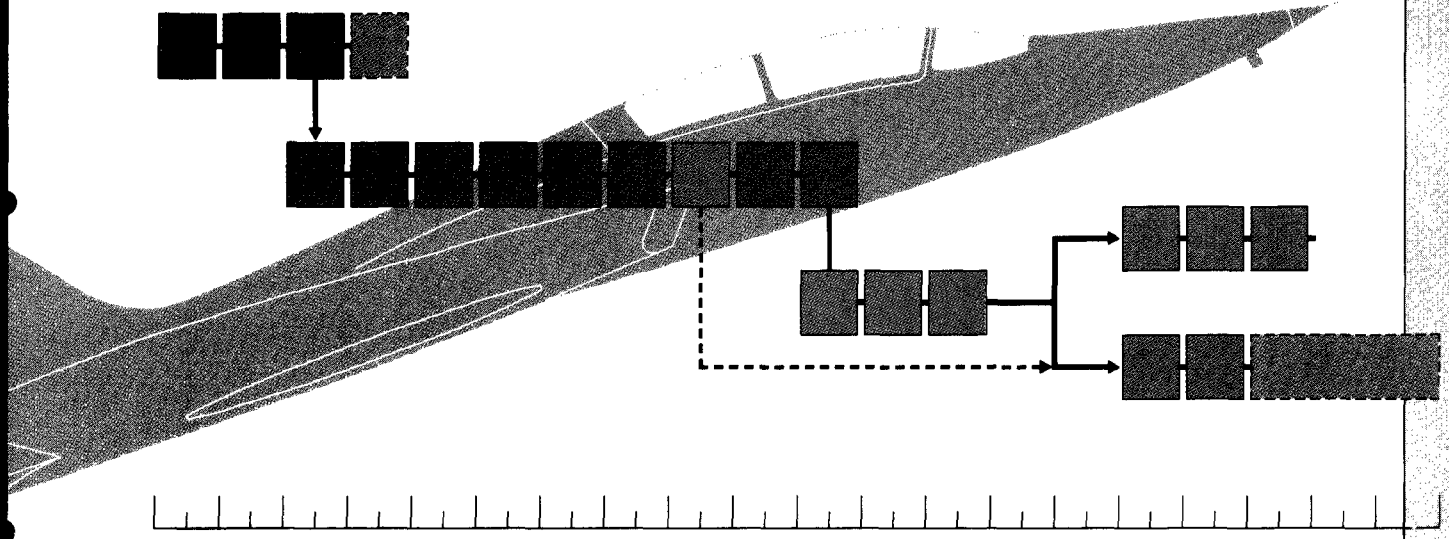






Diagnosis of telephone switching systems

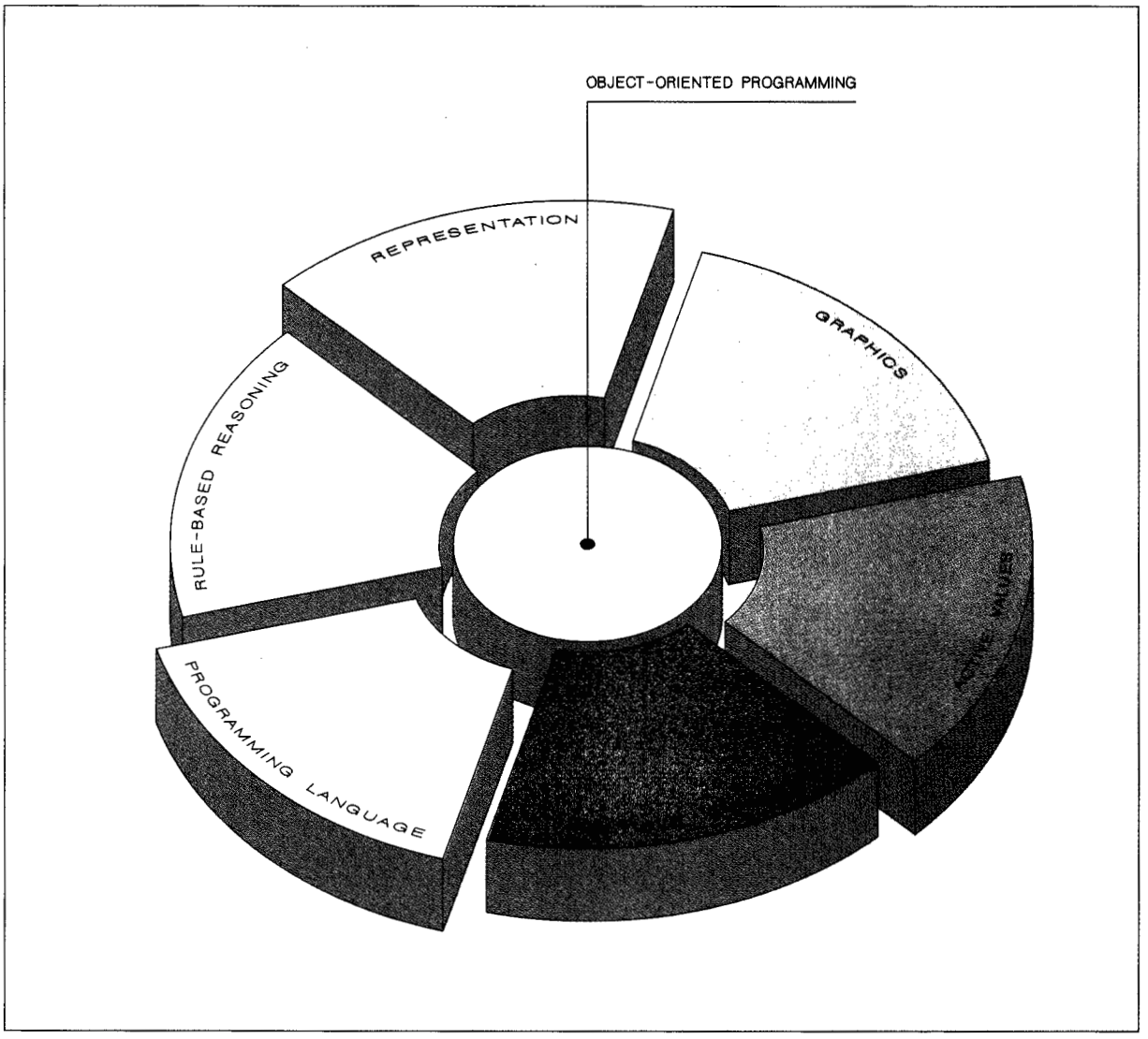
Telephone switches are extremely complex. A 20 000-line switch has some 10 000 circuit boards and up to 200 chips per board. Some built-in hardware fault detection circuitry exists, but considerable expertise is required to interpret the results of that fault data. The GTE Corporation developed a knowledge system to collect data, analyze faults, and suggest corrective action.



Generative Process Planner

Northrop Corporation developed a system which plans the sequence of steps required to manufacture a sheet metal part for an airplane. The system analyzes capabilities of machines and requirements for features in the parts, then plans sequences of machine steps which can create the required features.

Figure 2 Capabilities of the hybrid system KEE



A KEE Unit includes *slots*, which store the attributes of the Unit. Some slots may contain data and others may contain methods. The selection of slots and their uses give a KEE Unit its personality. Each slot, in turn, has its own attributes, which are called *facets*. Facets are used for many purposes. Facets can control the specific means by which inheritance is implemented for that particular slot. In some cases, local values override inherited values, and in other cases local values are appended to inherited values. KEE facets also provide a capability to establish the inheritance rules that govern the way a particular value may be derived from multiple ancestors. Facets are also used to store the type of information for the data. If a data slot contains a list of elements, for example, one facet can define the set from which elements may be drawn and another may control the minimum and maximum number of elements that may be included.

A pictorial representation of the functions offered by KEE is given in Figure 2. Object-oriented programming is at the center because it is viewed as the unifying concept of the program. The surrounding

elements represent major capabilities of a KEE system that are implemented by the KEE Units. Following are explanations of these elements.

Representation. When used for data storage, the KEE Unit structure is like a frame structure with multiple inheritance. There might be a general KEE Unit of financial instruments, for example, that defines a share of stock or a government bond. All financial instruments have certain characteristics in common. One is a current purchase price. Another is a term of duration, even if indefinite. Yet another is interest or dividend to be paid. However, different instruments may have important distinctions. In terms of risk and taxes, bonds differ from stocks. Government bonds are different from corporate bonds and so on. KEE Units can be formed to increase the ability to define specific attributes for each of the separate types of instruments.

Graphics can reveal the structure of an application and its behavior.

Rule-based reasoning. A KEE Unit may include a rule slot that forms a part of a forward-chaining or backward-chaining inferencing capability. Rules and facts in KEE are of the same form as those found in PROLOG. That is, they may be an assertion without condition: *This is a high-risk investment* is a fact. Or they may be an assertion with a condition: *If the recommended instrument is a common stock, the investment may involve high risk* is a rule. The rule-processing capability allows for the inclusion of variables. For example: *If x is a common stock, the investment has high risk.* This statement provides for resolution by unification. The rule system also allows recursion.

Programming language. A KEE Unit may have slots that store methods. In KEE, these methods are written in LISP or C, if available. Methods are initiated on the receipt of a message. As in Smalltalk, messages may be sent and received from other objects.

Active values. Sometimes called *demons* in other systems, active values are special methods that are initiated when an item of data within the KEE Unit is changed or accessed. Active values are useful in simulation and model-based reasoning. One might use active values to warn of an impending out-of-bounds condition, for example. They are also useful for simulating such user-interface devices as push buttons and switches.

Graphics. Graphics can reveal the structure of an application and its behavior, thus showing the users of an application that it is correct or visually revealing errors in applications that make them incorrect. The KEE system includes graphics explanation and debugging facilities, hard-copy graphics output, display of gages and meters, and a graphics programming language. A KEE Unit may contain a slot for a graphics representation or icon and automatically draws upon a hierarchy of functions available for graphics representation. The graphics possibilities include bit-mapped icons and line drawings. Also included are available functions such as overlaying, shrinking, zooming, and translation of position.

Multiple worlds. Multiple worlds is a facility in KEE that employs a truth-maintenance system for hypothetical reasoning. A user starts with a set of primitive facts and rules and, by proposing certain options—new rules or facts, follows them to a logical conclusion. This process may lead to a contradiction, which indicates that a particular set of facts and rules is inconsistent. On the other hand, the process may lead to the creation of sets of possible rules and facts that are consistent. This allows for a kind of "cut-and-fit" approach to problems to find a particular combination of conditions or solutions that work together.

Although KEE is an object-oriented development environment, it is not object-oriented in the sense of Smalltalk for several reasons. First, with KEE the user has ready access to the LISP language, which underlies the entire system. Thus it is not necessary to use the object-oriented programming facilities of KEE. Also, KEE encapsulates objects in a manner different from that of Smalltalk. In Smalltalk, access to an object's data is available only through methods (accessor functions) that are designed within the class. The KEE architecture allows access to values of data attributes of all objects. In other words, the scope of encapsulation in Smalltalk is circumscribed and the scope of encapsulation in KEE is broad.

In summary, KEE is a hybrid system, combining the power of a functional language, LISP, with object-oriented programming and with an environment that is rich in function for the knowledge-systems developer.

Clearly, one of the advantages of employing the object-oriented approach is the ability to pass the power of object-oriented programming to the user, along with a rich collection of capabilities. However, other advantages have accrued by this choice. KEE has been an evolving product, which means that change is constant, and new capabilities are being experimented with and added regularly.

For example, multiple worlds was not a part of the original KEE implementation. It was important to be able to integrate this new function within the original KEE framework. Further additions of function are planned, as well as the creation of new KEE systems for new machine architectures. Often, new personnel are asked to make these changes or additions, and it is believed that once the basic concepts of object-oriented programming are understood, the time required for a new employee to learn the system will be short, compared to a non-object-oriented approach. The ability to add this capability is a direct result of having already developed the basic KEE program, using object-oriented techniques.

In addition, KEEconnection™ is a recently developed software bridge between one or more relational databases that use the SQL query language and knowledge bases created within the KEE environment. Guided by mapping information supplied by the application developer, KEEconnection generates SQL queries and transforms the data obtained into slot values within KEE Units. It also provides the user with the ability to upload data, so that the database can incorporate the results of a knowledge-based analysis.

Status of object-oriented programming

Perhaps it is a problem of object-oriented programming that it can be viewed so many different ways, from a programming language to a programming style. It is perhaps best to view object-oriented programming as a discipline that can be employed with a wide range of implementations in any programming language. We have reviewed two approaches. Smalltalk implements all of the necessary and desirable characteristics of an object-oriented programming language except multiple inheritance. KEE is a hybrid language that implements most of the object-oriented programming features plus many additional functions.

How is object-oriented programming any better than the way things have previously been done? Present users of object-oriented programming see several benefits.

Organization. In his book *The Mythical Man Month,*[10] Fred Brooks makes clear that large programming projects are not infinitely divisible. One cannot reduce the elapsed time to completion by putting more people on the project. Object-oriented programming allows for a finer degree of subdivision, which is akin to the specialization of the workforce brought about by interchangeable parts. Since the idea of interchangeable parts has been around for more than three generations, few people can imagine what manufacturing might be like without it. Yet, there were some who said that the idea was unworkable because the need for closer tolerances would increase manufacturing cost. Over time, the concept of interchangeable parts had a profound effect on the ability to manufacture complex mechanisms.

Object-oriented programming allows a finer degree of subdivision.

To create self-contained component parts of a programming system and treat them independently is a goal similar to that of interchangeable parts. In object-oriented programming, each component is defined by its interface, and a complete description of the interface is all that is needed to be able to use a component successfully.

Reusable code. Over time, an extensive object library is developed. In doing so, new functions can borrow heavily on those that have preceded them, without the necessity of writing wholly new code. The hierarchical structure and inheritance capability allows for the creation of generic components that can be reused in many parts of the system.

Flexibility. The designer of each component has the freedom to make internal changes that do not affect the interfaces, such as the internal representation of data. Improvements that can be made within a class do not have to affect the users of that class. Thus there is less concern that unexpected problems will occur. New classes and methods can be added without affecting those already there, thus allowing for incremental modification. These benefits, if achieved, result in improved programming productivity and shortened development schedules.

Object-oriented programming has some drawbacks, however. It takes more computer processing to handle message passing than to perform function calls. Some users argue that the difference becomes minimal as the methods being performed become more complex. For simple methods that must be performed repeatedly, it is possible to program a "fast path" around the message-passing delays.

Also on the negative side is the increased training required. Whereas it is relatively easy to learn the basic concepts of object-oriented programming, it takes much longer for an individual to learn a large class library. Some argue that this is not a disadvantage, because the large library exists to provide capabilities that become powerful tools in the hands of the skilled user.

Some issues are not argued, but are merely questions the answers to which may depend upon the application or means of implementation. Inasmuch as object-oriented programming is an emerging technology, a number of these issues are being actively discussed, including the following.

Static versus dynamic binding. There is a question as to whether to define a particular symbol to storage (binding) at compile time or at run time. Advocates of compile-time static binding say that it helps to discover errors at that time, and the program ultimately runs faster. Others say that run-time dynamic binding frees the developer from the constraints of having to make such fixed decisions that may ultimately lead to more complex programs. A compromise position says that dynamic binding is best in the early development phases of a project, but that static binding is better later on, when the product is going to be installed for general use.

Static versus dynamic typing. The issue is similar to that of dynamic binding as to whether to determine a particular data element's type at compile time or run time.

Single versus multiple inheritance. Single inheritance derives from the classic tree hierarchy of classes, wherein each class has at most a single parent class. Multiple inheritance derives from the frame concept and argues that the tree structure is too limiting. Some objects can legitimately belong to more than one hierarchical structure. People, for example, might for some purposes be classified as men and women, and for other purposes as regular employees and temporary employees. It is not true that one form of inheritance is right and the other is wrong, but some languages today permit only a single inheritance path.

Interfacing with large databases. The basic concept of a database is contrary to the concept of an object. That is, the user of the data is not protected from changes in representation and must know how to use the data. An effective commercial system, however, is likely to use large databases and may also use a variety of programs and programming techniques with it. Initially, some of these programs may be object-oriented programs and others not. Therefore, there is a need for a bridge between the traditional view of data and the object view, which probably means that with object-oriented programming the data will be temporarily converted into an object form. Upon completion of processing, appropriate modifications will be made to the originating database, as KEEconnection provides. In the future, object capability may be added to database management systems to provide for some of the benefits of encapsulation.

Storage management. Storage management is the necessity to reclaim working storage that is no longer needed. LISP has a built-in "garbage collection" technique that periodically reclaims unused space. Smalltalk has methods for reclaiming space released by objects that no longer have references to them. In large systems, paging is the means of overlaying storage that has not been used for some time with new data that is likely to be used.

The basic question is whether storage management should be the responsibility of the object-oriented language, as in Smalltalk, or of the system. There may be some dependence upon whether the language being used employs static binding, where the system manages the storage, or dynamic binding, where the language polices its own storage use.

In the Objective C and C++ implementations, storage management is left to the responsibility of the system or the application programmer.

Concluding remarks

Object-oriented programming is one way of using the continually improving capabilities of computer technology to provide improvements in programmer productivity and user function. Object-oriented programming is being used for a wide range of applications, particularly for knowledge-based systems involving close human interaction and judgment. It may also be used as a tool for specifying and building large, integrated data processing systems.

This paper has reviewed Smalltalk as one example of a large system and the Knowledge Engineering Environment (KEE) as another, with emphasis on object-oriented programming as a vision and as a set of programming techniques. In our experience, the vision and the techniques have great value in programming as we build large systems that have long life cycles. We find the vision and the techniques immediately relevant to a central role in knowledge systems.

Cited references

1. A. Goldberg and D. Robson, *Smalltalk-80*, Addison-Wesley Publishing Co., Reading, MA (1983).
2. O. J. Dahl, B. Myhrhaug, and K. Nygaard, "The SIMULA67 Common Base Language," Publication S-2, Norwegian Computing Center, Oslo (May 1968).
3. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, Inc., Englewood Cliffs, NJ (1988).
4. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Co., Reading, MA (1986).
5. B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Co., Reading, MA (1986).
6. D. Weinreb and D. Moon, "Objects, message passing, and flavors," *LISP Machine Manual*, Symbolics, Inc. (July 1981).
7. D. Bobrow and M. Stefik, *The LOOPS Manual*, Xerox Corporation (1983).
8. R. Fikes and T. Kehler, "The role of frame-based representation in reasoning," *Communications of the ACM* **28,** No. 9, 904–920 (September 1985).
9. P. Wegner, "Dimensions of object-based language design," *OOPSLA '87 Conference Proceedings*, October 4-8, 1987, Orlando, FL, N. Meyrowitz, Editor; sponsored by the Association for Computing Machinery, 11 West 42nd Street, New York, NY (1987), pp. 168–182.
10. F. P. Brooks, Jr., *The Mythical Man Month*, Addison-Wesley Publishing Co., Reading, MA (1979).

Richard P. Ten Dyke *P.O. Box 789, Bedford, New York 10506.* Mr. Ten Dyke is an independent consultant working in the field of advanced application technology. Prior to his retirement from IBM, he was the assistant for business analysis for Advanced Systems, Enterprise Systems, where he worked in the Artificial Intelligence Project office. He holds a B.S. degree from the University of Minnesota, and an M.B.A. from Harvard University.

John Kunz *IntelliCorp, 1975 El Camino Real West, Mountain View, California 94040.* Dr. Kunz is chief knowledge-systems engineer and director of manufacturing applications at IntelliCorp. He is one of the initial developers of the IntelliCorp KEE system and has since developed applications systems in diverse areas including experiment design for molecular biologists, power plant control, petroleum exploration, project management, and factory scheduling. Prior to joining IntelliCorp, he directed development of the PUFF medical diagnosis system, the first Artificial Intelligence-based system to be used routinely. Author of numerous articles and book chapters, he has also spoken to many groups on Artificial Intelligence and its applications. Originally trained in engineering and computer science at Dartmouth and UCLA, his Ph.D. emphasized computer science and physiology at Stanford.

Reprint Order No. G321-5370.