Guest Editors’ Introduction:

Object-Oriented Computing

David C. Rine, George Mason University
Bharat Bhargava, Purdue University

Problem-solving is the fundamental activity of people working in science and engineering. It has two aspects: process and structure. The process begins with a problem statement (set of requirements) and ends with a solution statement (set of specifications). Of course, the solution statement may be further realized in some tangible product.

The process is generally called reasoning, and the result of reasoning is structure. Solution statements take on structure so that we can understand the result of reasoning.

We consider three kinds of reasoning here. Two are traditional: Induction, reasoning from particulars to generals, is common in science and inductive mathematics; deduction, reasoning by inference from premises, is common in software and deductive mathematics. The third kind, eductive reasoning, is newer. It brings out new, kinetic features from previously potential or latent features. Eduction is common in system maintenance, where new system requirements — previously hidden or not yet brought out — are incorporated into the system.

We discuss two ways of representing structure: function and form. We can represent function by codifying behavior, and we can represent form by classifying features. So one result of reasoning or thinking about a problem is a set of “rules” of behavior, and another result is a set of “classes” of features.

A way of thinking about something is sometimes called a paradigm. Hence, the result of reasoning about function is called the rule-based paradigm, and the result of reasoning about form is called the class-based paradigm or, because classes are composed of objects, the object-oriented paradigm.

Traditionally, science seems more interested in form and engineering more interested in function. However, both disciplines are interested in what happens to their structures, whether form or function, when further reasoning, inductive or
Object-oriented systems development

Object-oriented systems are as old as the record of human thought. Genesis organizes the Creation and presents the first genealogies. Early Greek writing distinguishes phusis — the kind, or natural constitution of emphasis — from genos — the hierarchy of ancestors and descendants.

The following outline of object-oriented systems development in science and computing includes bibliographies of further reading.

I. Historical background
   A. Object-oriented structures in science
      1. Theories and applications of classifications
      2. Knowledge representation strategies
   B. Early object-oriented thought in computing
      1. Programming environments (for example, Simula and Smalltalk)
      2. Artificial intelligence (for example, concept formulations)
      3. Software systems design (for example, modularity and abstraction)
   Further reading

II. Recent developments in object-oriented analysis and design
   A. Design methods (for example, software engineering methods)
   B. Programming environments (for example, user interfaces, object management systems, dynamically typed languages, statically typed languages, and distributed systems)
   C. Domain modeling (for example, domain analysis and modeling)
   D. System development metrics (for example, cost models and reliability models)
   E. Management approaches (for example, contractual representations)
   Further reading

III. New disciplines using object-oriented thought
   A. Economics (for example, property rights and agoric open systems)
   B. Software business (for example, management constrained design)
   C. Legal aspects (for example, law-governed and contractual design)
   Further reading

IV. Progress in performance and user friendliness of object-oriented computing
   A. Speed and memory considerations
   B. Tools
   Further reading
More recent thought

Software and knowledge engineering. The rapid development of computer technology has engendered various disciplines of computer engineering. Among them are software engineering and knowledge engineering, which can also be thought of as experience-based and intelligence-based engineering. While a software engineer and a knowledge engineer might seem to talk about completely different subjects, they come together on, for example, issues involving software maintenance. 2

Software engineering addresses the technological aspects of developing high-quality software products within budget and on time. There have been at least two software engineering "revolutions": structured programming in the 1970s and object-oriented programming in the 1980s. 4 Issues in software engineering usually center on the life cycle model. Simply speaking, the life cycle model consists of only two parts: development and maintenance.

Knowledge engineering is a newer term than software engineering. It was coined in the mid-1970s, about 10 years after software engineering was. It specifically refers to the process of putting an expert's nonalgorithmic, empirical knowledge, including experiences and intelligence, into an expert system. 5 Its engineering issues are often discussed under such topics as knowledge acquisition and knowledge representation. An expert system can be implemented in many ways, using conventional or unconventional paradigms. An expert system implemented in a conventional paradigm differs little from a conventional system, since everything is coded or represented procedurally, even though the way the knowledge is acquired might differ.

Knowledge engineering first used rule-based paradigms. The RBP refers to the production system architecture. 6

Example of an object-oriented world view and model

Consider a world view where people are thought of as objects. In this view, a People object has

1. Ideas, facts, knowledge, and sentences.
2. Methods for processing facts and sentences such as understanding, reducing, changing, and remembering.
3. Communication with other objects through messages (for example, Remember when <sentence phrase>); the messages may or may not be understood.

People objects can be placed in a "situation" that uses certain kinds of sentences and methods. The situation may involve an entire class of People objects. However, through learning and experience, each People object has inherited other sentences and methods for processing them and may therefore be able to deal with other unexpected situations (for example, a change in the topic of conversation from computer science to basketball).

People objects may be combined and integrated into classes for such purposes as carrying on a conversation in some domain, performing a task involving teamwork, having parties, or playing games involving different teams. These activities are called object-oriented development. If object-oriented development is carried out using a predefined "metalanguage" of instructions, then the activities are called object-oriented programming.

People objects use their native ability to deal with ideas, facts, knowledge, and sentences in imprecise and abstract ways. People objects can process ideas and messages by hiding much of the implied information associated with these ideas and messages. People objects can send, receive, and adapt to ideas and messages dynamically in real time.

These last three statements are associated with the three technical terms abstraction, information hiding, and dynamic binding.

People-object activities are represented in the computer by

1. Analyzing the People objects and their activities.
2. Developing ways to represent People objects in the computer — their ideas and methods — while using the features of information hiding, data abstraction, dynamic binding, and inheritance.
3. Developing design techniques for classifying and integrating the computer-represented People objects in productive ways.

Finite state automata and formal logic can be used to represent People objects mathematically.
A system implemented in the RBP is called a rule-based system. An RBS typically consists of a database, a rule base, and a generic inference engine. The database is a collection of data reflecting the status of the application domain. The rule base consists of a set of rules — the knowledge (experiences, intelligence) extracted from a domain expert and coded declaratively. The domain-independent inference engine controls the entire system by deciding which rule to apply under a particular circumstance.

The RBP has the theoretical advantage of making no distinction between development and maintenance, since both involve adding, deleting, or modifying rules. Unfortunately, reality has failed to confirm this advantage. Experience shows that RBSs are at least as difficult to maintain as many conventional systems. For one thing, as Davis* observed early on, RBSs are limited by the requirement for a "knowledge czar." This limitation is imposed by the need to avoid conflicts that might result from different experts' points of view. One RBS — or more specifically, one rule base — should incorporate the expertise of just one expert. This means that the quantity and quality of the expert's knowledge largely determines RBS performance.

Practically, this limitation implies that only one knowledge engineer should be responsible for the construction and maintenance of a rule base. This limitation is fatal from the software engineering point of view: The developer is usually not the maintainer, and many people are involved over the entire RBS life cycle.

There are very few design issues in developing a single RBS. Moreover, the RBP has failed to provide mechanisms for the successful evolution of an RBS over a reasonable period of time. Meta-rules were introduced as a control mechanism to compensate for this inadequacy. Among their drawbacks, however, is their transparency, which makes the system hard to test and maintain.

Many researchers saw the limitations of existing expert-system technology, particularly the RBP, early on. Unless the RBP is abandoned completely, software engineering must be extended to address the problems it raises.

Object-oriented programming. In contrast to the RBP, the object-orient-

---

**Professional meetings and journals**

Many international conferences and journals have been formed in response to the revolution of object-oriented thought in the sciences and engineering. For example, the International Conference on the Technology of Object-Oriented Languages and Systems (TOOLS) meets twice each year, and the International Conference on Object-Oriented Programming Systems and Languages (OOPSLA) meets annually. These conferences attract thousands of participants, excellent papers, and many exhibits.

Moreover, some general software engineering conferences, such as the International Conference on Software Engineering and the European Conference on Software Engineering, both held annually, attract many papers on object-oriented thought. There are also many specialty symposia and workshops each year, such as the Symposium on Object-Oriented Methodologies for the Earth Sciences, which took place in 1991.

Periodicals specializing in object-oriented issues include the *Journal of Object-Oriented Programming*.

---

**Terminology**

**Object** — an abstraction of a real-world entity.

**Class** — a set or collection of objects having common features.

**Feature** — a routine, activity, or attribute serving as part of the definition of an object or class.

**Method** — a feature performing an operation on an object.

**Instance variable** — an attribute feature of an object.

**Execution-time creation** — the dynamic or runtime creation of an object or instance of a class.

**Message** — a protocol composed of a method or routine and an object reference or address.

**Object-oriented development** — construction of an object-oriented system from its requirements by object-oriented analysis, design, and programming.

**Object-oriented language** — a well-defined notation that supports object-oriented properties and specification of an object-oriented system.

**Object-oriented programming** — an object-oriented development method that leads to a software system based on the objects every system/sub-system manipulates, rather than the function it is meant to ensure.

**Object-oriented properties**

**Information hiding** — separation of representation details of an object, class, or system from its application domain details.

**Abstraction** — separation of unnecessary details from systems requirements or specification so as to reduce complexities of understanding requirements or specification.

**Dynamic binding** — instantiation of an identifier or variable, defined by use of an object-oriented language, with an object during the execution of a system.

**Inheritance** — a relationship between two classes of objects such that one of the classes, the child, takes on all relevant features of the other class, the parent.
ed paradigm (OOP) is a general-purpose programming paradigm that works for a wide variety of applications. Like the RBP, the OOP can be viewed as revolutionary, but it is perhaps more appropriately considered evolutionary. As a programming paradigm, the existing OOP is perhaps more procedural than declarative. This is because the sharing and reuse of attributes and data-manipulation routines among objects is a key issue in OOP, and the implementation of these routines through "inheritance" is often no different from the implementation in conventional programming. Inheritance is a relation between objects or classes of objects. It allows an object or class to share or reuse resources previously developed for another object or class.

OOP is inclusive, just as structured programming was two decades ago. It differs, however, from structured programming's traditional association with functional design methods such as functional decomposition, dataflow diagrams, or data structure design. In OOP, objects are first categorized into classes and organized hierarchically according to their dependency and similarity. Each class comprises a set of attributes reflecting the objects' generally static properties and a set of routines (in Smalltalk, methods) that manipulate these attributes. Then relations between classes, such as inheritance, are designed.

Our experience leads us to advocate OOP as the best programming paradigm available for implementing certain types of systems, but we are not proposing OOP as a panacea for all the problems of software maintenance, reuse, reliability, and quality. In fact, software engineering has no "silver bullet." As applications evolve quickly, the challenges to maintain software systems increase just as fast. Software maintenance has become a dominant issue in knowledge as well as in software engineering. The RBP has many problems that can be fatal, and appropriate applications for this paradigm are probably very few in the real world.

On the other hand, the OOP may be the best programming paradigm available for implementing certain types of systems, such as expert systems, since the OOP conforms to conventional wisdom. We see that development and maintenance of expanded intelligent management information systems require adapting to rapid and significant changes in the application. We have also seen adaptation and maintenance to be major stumbling blocks for enterprises with which we work. We therefore propose using the OOP, as opposed to the RBP, for intelligent systems.

References


David C. Rine is a professor of computer science as well as information and software systems engineering at George Mason University, and a senior researcher in the university’s Center for Software Systems Engineering. At CSSE, his research has focused on object-oriented development, software maintenance and reuse, and software development environments.

Rine received his PhD in mathematical sciences in 1970 from the University of Iowa. He has published more than 140 papers and received numerous awards including the IEEE Centennial Award, the IEEE Computer Society's Pioneer Award, Meritorious Service Award, and Special Award. He has coauthored several computer texts and is presently associate technical editor for Computer Journal, area editor for object-oriented computing, and editor for the IEEE Computer Society's Readings in Computer Science and Engineering series.

Bharat Bhargava is a professor of computer science at Purdue University. His research involves both theoretical and experimental studies in distributed database systems. His research interests include adaptability in distributed systems, replication management, and new paradigms in communications for high-performance transaction processing. Bhargava is the editor of Concurrency Control and Reliability in Distributed Systems, published by Van Nostrand and Reinhold in 1987. At the 1988 IEEE Data Engineering Conference, he and John Riedl received the best paper award for "A Model for Adaptable Systems for Transaction Processing." Bhargava is on the editorial board of IEEE Transactions on Knowledge and Data Engineering. He is editor of the newsletter for the IEEE Computer Society's Technical Committee on Distributed Processing.

Readers can contact David C. Rine at the Department of Computer Science, Science and Technology Bldg. II, Rm. 3.45, George Mason University, Fairfax, VA 22030; e-mail drine@gmu.cacx.edu.