set. Although we have had some tools for building specific components for a number of years (for example, compiler compilers), we are only beginning to develop "meta tools" for automating the construction of complex, integrated development environments.

By "tools," I do not simply mean entities such as specification languages, syntax-directed editors, and symbolic debuggers, but also the formal mechanisms that we employ as well. We are undertaking the slow, costly evolution from the current "arts and crafts" orientation of software development to one that is more scientifically based because we believe that a formal approach to the specification, design, prototyping, implementation, and verification processes is necessary if we are to improve each process, construct reliable and efficient software, and have higher software productivity. I do not argue for formalism in the lockstep, implementation sense, but endorse formal methods that shed light on the real problems we must resolve. Therefore, an important part of an "integrated tool set" is its formal components. That is why I welcome articles that bridge the applied-formal tools gap.

There are often many different principles or methods for using tools. If we follow the principle of "reusability" when implementing a system, for example, we attempt to develop as many subsystems as we can so they can be "reused" in other parts of the specific system. If we can do this in such a way that the subsystems can be reused in other systems as well, either in a prototype or production context, we will have leveraged our efforts significantly. Can we generalize the principle of reusability to the subparts of the specification, design, prototype, validation, and documentation of a system as well? Can we develop methodologies (integrated groupings of formal and applied principles and methods) for building systems? Can methodologies span application domains—for example, are the methodologies for building expert systems the same as those for building real-time process control systems, or transaction-oriented commercial systems, or avionics communication systems? It would be useful for those of you who have had some experiences in this area to share them with the general readership.

If we are going to provide aids to help automate the software development process, we must understand not only the process, but also the entities that are produced. We must model both. Such modeling has been useful in providing a number of results, from the chief-programmer-team concept and code reviews in the "process" camp to the queuing models of operating systems usage and the data dependency graph models for optimizing compilers in the "entity" camp. The basic thesis is that in order to improve the present software development process and the resultant entities, we must first understand them. We can then propose changes to this process, to the formal and applied tools we employ, and to the entities. Then, after implementing these changes, we can evaluate them. "Software" behavior models may lead to methods to automatically reorganize programs for distributed and/or real-time applications. Some investigations in alternative software development environments are being undertaken, and we hope to report on them in our magazine.

One of the challenges for IEEE Software in 1985 and beyond will be to continue to provide a timely environment for the presentation of articles on these and related problems in the software domain. We welcome your participation in this endeavor.

Bruce D. Shriver
Editor-in-Chief