REPRINTED FROM IBM SYSTEMS JOURNAL, VOL19, NO 4, 1980; © 1980, 1999

Software engineering may be defined as the systematic design and development of software products and the management of the software process.

The general principles of software engineering are set forth in Part I, in which the author relates software engineering to the whole field of the system development process—system engineering, hardware engineering, software engineering, and system integration. Presented briefly are overviews of the major aspects of software engineering—design, development, and management.

The management of software engineering Part I: Principles of software engineering

by H. D. Mills

In the past 20 years, the Federal Systems Division of the IBM Corporation has been involved with some of the nation's most complex and demanding software developments. These include the ground support software for the NASA Manned Space Series of the Mercury, Gemini, Apollo, and Skylab Programs (reaching the moon with Apollo), and both the ground and space software for the NASA Space Shuttle Program. FSD has also developed software for the Safeguard Anti-Ballistic Missile System, for the En-route Traffic Control System for the FAA, and many other major civil and defense systems.

Software engineering began to emerge in FSD some ten years ago in a continuing evolution that is still underway. Ten years ago general management expected the worst from software projects—cost overruns, late deliveries, unreliable and incomplete software. Today, management has learned to expect on-time, within-budget deliveries of high-quality software. A Navy helicopter/ship system, called LAMPS, provides a recent example. LAMPS software was a four-year project of over 200 person-years of effort, developing over three million and integrating over seven million words of program and data for eight different processors distributed between a helicopter and a ship, in 45 incremental deliveries. Every one of those deliveries was on time and under budget. A more extended example can be found in the NASA space program, where in the past ten years, FSD has managed some 7000 person-years of software development, developing and integrating over a hundred million bytes of program and data for ground and space processors in over a dozen projects. There were few late or overrun deliveries in that decade, and none at all in the past four years.

There have been two evolutions in FSD: first, an evolution in ideas, leading to a growing discipline in both the management and technical sides of software engineering, and second, an evolution

 0018-8670/99/$5.00 © 1999 IBM

in the number and skill of people using the discipline. This evolution has not been without pain and attrition. Software is a new subject of human endeavor. Just as programming has evolved from a cut and try individual activity to a precision design process in structured programming, software engineering has evolved from an undependable group activity to an orderly and manageable activity for meeting schedules and budgets with high-quality products.

It is one thing to talk about orderly software development, and quite another to achieve it. The basis for this orderly control is mathematical discipline, even though the problem being solved by the software may not be mathematical. The key management standards of software engineering in FSD are based on mathematical theorems about how programs can be structured, documented, and organized into larger systems, because without theorems for bedrock, choices reduce to matters of management style and individual experience.

The FSD Software Engineering Education which supports the Program is highly mathematical for both managers and programmers. Set theory, logic, mathematical functions, and state machines play key roles in education, not for the sake of mathematics itself, but because practical experience has shown that that level of precision is required in order to do more than talk about orderly software development.

What is software?

Software began as a synonym for computer programs, but the term has taken on a much more extensive meaning. The effective use of computer hardware requires more than programs. It requires well-informed users and human procedures for computer operations, data entry, and program execution. These requirements call for instructions for humans of no less precision and completeness than programs for the computers. Thus, operators' guides, users' guides, etc. become as important to a system operation as programs. Further, the users must understand well enough what the computers do to correctly interpret their outputs and intelligently prepare their inputs to meet operational objectives. Thus, requirements and specifications of computer programs and systems are of vital importance to the users as well.

Although computers began as single units serving a single user at a time, the rapid growth of multi/distributed processing systems to serve multi/distributed users has greatly expanded the role of software. Software is the logical glue that can hold many computers and digital devices of all kinds together in a coherent system, which in turn interacts with many kinds of people—clerical, pro-

fessional, staff specialists, and management—in the operation of an enterprise.

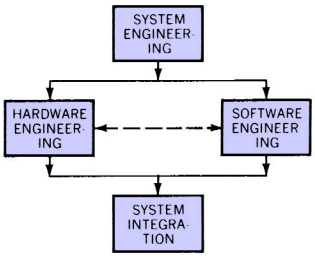
As a result of the pervasive role of software in a multi/distributed processing system, it seems proper to redefine the term software from its usual meaning of single programs to mean logical doctrine for the harmonious cooperation of a system of people and machines—usually many kinds of people and many kinds of machines. In such a system, the agents of action are people and machines, with the blueprints for their action supplied by software. A human procedure is as important to the system as a machine procedure. People have radically different instruction sets than machines, including an operation called "use your common sense," but they have instruction sets just the same. The synchronization of two people or a person and a machine is as important as the synchronization of two machines, but people often supply self-synchronization capabilities. Even "off the shelf machines" have an analog in "people with presently available skills."

Thus, software consists of operational requirements for a system, its specifications, design, and programs, all its user manuals and guides, and its maintenance documentation. Further, this whole software complex needs to evolve as a consistent whole as the operation evolves, as new hardware is added, and as new people are added. That is, software is typically a set of logical blueprints for the operation and use of a multi/distributed processing system by an organization of people in its natural evolution over time.

What is software engineering?

Software engineering is a growing set of disciplines and procedures for the dependable development and maintenance of software, as embodied in the FSD Software Engineering Practices, and discussed in Reference 1. For a wider perspective, we can identify the following four definite functions in an overall system development process, the relationships among which are illustrated in Figure 1.

Figure 1 System development



Software engineering stands between system engineering and system integration, accepting from system engineering the system software requirements and resources, and providing system integration with the software for meeting those requirements with those resources. Thus the total software of a system is a joint product of system engineering and software engineering, which begins with a defined system purpose and a defined configuration of hardware.

Of course, operating systems, compilers, and programming support systems all represent special and specialized software sys-

tem developments, and the disciplines and procedures of software engineering apply fully to them. But we are usually more preoccupied with application systems, which make use of such support systems as extensions of the hardware.

The FSD practices classify the disciplines of Software Engineering into the following three categories:

- *Design*—system design, module design, program design, and data design, all of which culminate in source code in one or more compilable programming languages, as well as in linkage editor, loader, and job control languages.
- *Development*—organization of design activities into sustained software development, selection, and control of design support facilities, code management, test, and software integration planning and control.
- *Management*—work breakdown and organization procedures, estimation, and scheduling of personnel and computer resources required for software design and development, measurement and control of software design and development.

Software engineering design

Attention to the principles of software design has focused on three distinct areas during the past decade and has resulted in an abundance of useful and well-tested material on the following subjects:

- Sequential process control—characterized by structured programming and program correctness ideas of Dahl, Dijkstra, and Hoare,[2] Hoare,[3] Linger, Mills, and Witt,[4] and Wirth.[5,6]
- System and data structuring—characterized by modular decomposition ideas of Dahl, Dijkstra, and Hoare,[2] Ferrentino and Mills,[7,8] and Parnas.[9].
- Real-time and multiple/distributed processing control—characterized by concurrent processing and process synchronization ideas of Brinch Hansen,[10] Hoare,[11] and Wirth.[12]

Software design requires the integration of these three areas into a systematic process, as discussed in Reference 13. These design principles provide increased discipline and repeatability for the design process. Designers can understand, evaluate, and criticize each other's work in a common, objective framework. As pointed out by Weinberg,[14] people can better practice egoless software design by focusing criticisms on the design and not on the author. These design principles also establish the criteria for more formalized design inspection procedures that permit designers, in-

spectors, and management to better prepare, conduct, and interpret the results of periodic design inspections.

Software engineering development

Although the primary thrust of software engineering is embodied in design, the organization and support of design activities into sustained software development is an equally important activity, as discussed in References 1, 15, and 16. The selection of design and programming languages and their support tools, the use of library support systems to maintain and monitor a design under development, and the implementation of a test and integration strategy will all affect the design process in major ways. The disciplines and procedures needed to sustain software development must be scrutinized and chosen as carefully as design principles.

Intellectual control is the key to orderly software development. It is made possible by a sequence of logically equivalent software descriptions, beginning with high-level specifications and proceeding through successively lower-level specification refinements until the level of source code is reached. Successive descriptions can be baselined and validated to milestones, so that the intermediate progress of software development is more visible to management. This activity of creating a sequence of more and more detailed specification refinements of an initial specification is the process of *top-down development*.

The intellectual control and management of design abstractions and details is the basis for the development discipline. Design and programming languages are required that can deal with procedure abstractions and data abstractions, with system structure, and with the harmonious cooperation of multi/distributed processes. Library support systems are required that can handle the convenient creation, storage, retrieval, and correction of design units, and provide the overall assessment of design status and progress against objectives.

The first guarantee of quality in design is in well-informed, well-educated, and well-motivated designers. Quality must be built into designs, and cannot be inspected in or tested in. Nevertheless, any prudent development process verifies quality through inspection and testing. Inspection by peers in design, by users or surrogates, by other financial specialists concerned with cost, reliability, or maintainability not only increases confidence in the design at hand, but also provides designers with valuable lessons and insights to be applied to future designs. The very fact that designs face inspections motivates even the most conscientious designers to greater care, deeper simplicities, and more precision in their work.

Software engineering management

Management from a software engineering viewpoint is primarily the management of a design process, and represents an equally difficult intellectual activity. While the process is highly creative, it must still be estimated and scheduled, so that the various parts of the design activity can be coordinated and integrated into a harmonious result, and so that users and other functions of system development can plan on this result. The intellectual control that comes from well-conceived design and development disciplines and procedures is invaluable in this process. Without that intellectual control, even the best managers face hopeless odds in trying to see the work through.

To meet cost/schedule commitments based on imperfect estimation techniques, a software engineering manager must adopt a manage-and-design-to-cost/schedule process. That process requires a continuous and relentless rectification of design objectives with the cost/schedule needed to achieve those objectives. Occasionally, a brilliant new approach or technique which increases productivity and shortens time in the development process may simplify this. But usually, the best possible approaches and techniques have already been planned, and a shortfall or windfall in achievable software requires consultation with the user to make the best choices among function, performance, cost, and schedule. The intellectual control of software design not only allows better choices in a current development, but also stimulates subsequent improvements in function or performance for a well-designed baseline system resulting from the current development.

In software engineering, there are two parts to an estimate—making a good estimate and making the estimate good. The software engineering manager must see that both parts are right in addition to ensuring the right function and performance. That is not an easy task and never will be, but there is help on the way, as described in the companion articles and in the references.

ACKNOWLEDGMENTS

The authors thank FSD President John B. Jackson for giving them as well as other developers and students of the software engineering program the leadership and means to implement this program. We also thank James A. Bitonti for setting for us the goal of developing a written base of procedures for the educational program and project compliance accountability.

The author is located at the IBM Federal Systems Division, 10215 Fernwood Road, Bethesda, MD 20034.

CITED REFERENCES

1. H. D. Mills, "Software development," *IEEE Transactions on Software Engineering* **SE-2,** No. 4, 265–273 (December 1976).
2. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, Inc., New York (1972).
3. C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM* **12,** No. 10, 576–583 (October 1969).
4. R. C. Linger, H. D. Mills, and B. L. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Inc., Reading, MA (1979).
5. N. Wirth, *Systematic Programming: An Introduction*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1976).
6. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1973).
7. A. B. Ferrentino and H. D. Mills, "State machines and their semantics in software engineering," *Proceedings of IEEE Comsac '77*, IEEE Catalog No. 77Ch1291-4C, 242–251, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854 (1977).
8. H. D. Mills, *On the development of systems of people and machines*, Springer-Verlag, New York (1975).
9. D. L. Parnas, "The use of precise specifications in the development of software," *Proceedings of IFIP Congress 77, Toronto, August 8–12, 1977*, B. Gilchrest, Editor, North-Holland Publishing Co., New York (1977), pp. 861–867.
10. P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1977).
11. C. A. R. Hoare, "Monitors: An operating system structure concept," *Communications of the ACM* **17,** No. 10, 549–557 (October 1974); "Corrigendum," *Communications of the ACM* **18,** No. 2, 95 (February 1975).
12. N. Wirth, "Toward a discipline of real-time programming," *Communications of the ACM* **20,** No. 8, 577–583 (August 1977).
13. H. D. Mills, "Software engineering," *Science* **195,** No. 4283, 1149–1205 (March 18, 1977).
14. G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold Co., New York (1971).
15. F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal* **11,** No. 1, 56–73 (1972).
16. M. A. Jackson, *Principles of Program Design*, Academic Press, Inc., New York (1975).

Harlan D. Mills

Federal Systems Division, Bethesda, Maryland

Dr. Mills has been employed by the IBM Corporation since 1964. He received an Outstanding Contribution Award in 1973 for new programming methodologies, including top-down program design, techniques used for structured programming, and the Chief Programmer Team concept. Dr. Mills was named an IBM Fellow at that time. He served on the Corporate Technical Committee in 1973–74. He was director of software engineering and technology in the Federal Systems Division. Dr. Mills received a Ph.D. in mathematics in 1952 from Iowa State University. He was named an Honorary Fellow by Wesleyan University in 1962, and a Fellow of the Association of Computer Programmers and Analysts in 1975. He has served on faculties of the Iowa State University, Princeton University, New York University, and The Johns Hopkins University. Dr. Mills has been Adjunct Professor of Computer Science at the University of Maryland since 1975.