Leading and Directing

Because software development is a labor-intensive (intellect-intensive) activity conducted by teams of individuals, issues of leadership, motivation, and team building are of paramount importance to successful software project management. Leadership and management are related but distinct concepts. Management involves presiding over institutional processes, allocating resources and assigning responsibilities, and checking up on work assignments. Leadership, on the other hand, is concerned with issues such as organizational politics, vision, communication and coordination, values, motivation, and improving the work processes and work structures.

A particularly difficult issue for many software managers is delegation of technical decisions to those best qualified to make them. Every software development project should have a chief architect (i.e. lead engineer, senior designer). On small projects (three to five people), it may be possible (and desirable) for the manager to also play the role of chief architect. On larger projects, the manager and chief architect should be different persons because of the amount of work to be done and the different skills needed for managerial leadership and technical leadership.

Motivation is the drive to satisfy one's psychological needs. Studies of motivational factors for software developers have shown that the major motivators are autonomy, professional growth, and confidence in the technical leadership. Autonomy refers to the freedom to make decisions concerning the best way to do one's job. Professional growth is the opportunity to learn and apply new skills. Confidence in technical leadership refers to the confidence that one's technical skills and energy are being used to best effect. It is the leader's responsibility to create a work environment in which software developers can fulfill their psychological work needs and thus obtain a sense of job satisfaction.

Software systems are, for the most part, developed by teams rather than by individuals. It is therefore essential that the leader/manager knows how to build an effective team so that software developers can work in a cooperative manner toward shared common goals.

The tasks of a leader/manager are thus to communicate values, vision, and day-to-day information; to delegate technical decisions to those best qualified to make them; to provide a work environment in which individuals can satisfy their psychological needs; and to develop effective work teams. These tasks are particularly difficult for software managers who are promoted into management as a reward for their technical skills, rather than their "people" skills, and who may have received no training in leadership or team building. The text *Peopleware* (DeMarco, 1989) provides insightful details concerning the leading and directing of software engineers.

Representing the Project

The project manager is the project representative and spokesperson. Different stakeholders and constituencies have different needs for information concerning and involvement in a project. Stakeholders and constituencies for a software project include (perhaps) higher-level managers, supporting departments such as configuration management and quality assurance, subcontractors, affiliated projects (both hardware and software), affiliated contractors, customers, various types of users, and the system acquirer.

Bibliography

1971. Weinberg, G. *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold.
1981. Boehm, B. *Software Engineering Economics.* Upper Saddle River, NJ: Prentice Hall.
1982. DeMarco, T. *Controlling Software Projects.* New York: Yourdon Press.
1985. Fairley, R. *Software Engineering Concepts.* New York: McGraw-Hill.
1989. Boehm, B. W. *IEEE Tutorial on Software Risk Management.* Los Alamitos, CA: IEEE Computer Society Press.
1989. DeMarco, T. *Peopleware.* New York: Yourdon Press.
1995. Brooks, F. *The Mythical Man-Month,* Rev. Ed. Reading, MA: Addison-Wesley
1997. Thayer, R. (ed.) *IEEE Tutorial on Software Engineering Project Management.* Los Alamitos, CA: IEEE Computer Society Press.
1998. Pfleeger, S. *Software Engineering Theory and Practice.* Upper Saddle River, NJ: Prentice Hall.
1998. *IEEE Std 1058, 1998 Edition; Standard for Software Project Management Plans.* Piscataway, NJ: The Institute of Electrical and Electronics Engineers, Inc.

Richard E. Fairley

SOFTWARE PROTOTYPING

For articles on related subjects *see* SOFTWARE ENGINEERING; SOFTWARE PROJECT MANAGEMENT; SOFTWARE REUSABILITY; and SOFTWARE TESTING.

A *software prototype* is an executable model of a proposed software system that accurately reflects chosen aspects of the system, such as display formats, the values computed, or response times. Software prototyping is an approach to software development that uses prototypes to help both the developers and their customers visualize the proposed system and predict its properties in an iterative process as shown in Fig. 1.

Prototypes are used extensively by designers and engineers working in other disciplines. For example, architects build scale models of buildings to aid visualization

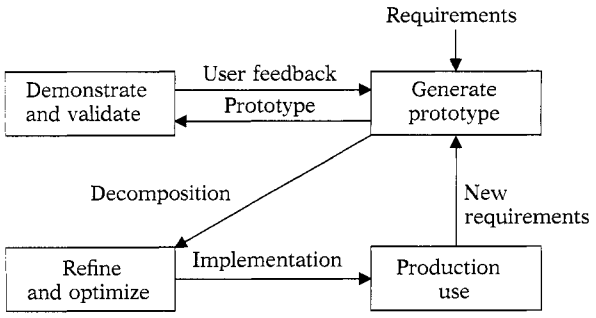

Figure 1. The software prototyping process.

of three-dimensional relationships, aeronautical engineers build scale models of airplanes to measure lift and drag in wind tunnel tests, and electrical engineers build breadboard circuits to check the validity of designs based on simplified ideal models of physical components. The common purpose of these prototypes is to reduce the uncertainty about the properties of a proposed design before it is implemented.

The main incentive for using prototypes is economic: scale models and prototype versions of most systems are much less expensive to build than the final versions. Prototypes should therefore be used to evaluate proposed systems if acceptance by the customer is in doubt. The motivation for software prototyping is essentially the same, and has become more urgent as systems being developed have grown more complex, and hence more expensive and more likely to have requirements errors (errors in specifying the system).

Software prototypes may not satisfy all of the constraints on the final version of the system. For example, a prototype may provide only a subset of all the required functions, it may be expressed in a more powerful or more flexible language than the final version, it may run on a machine with more resources than the proposed target architecture, it may be less efficient in both time and space than the final version, it may have limited capacity (databases may be implemented in main memory), it may not include facilities for error checking and fault tolerance, and it may not have the same degree of concurrency as the final version.

It is generally not possible to put a prototype into production use. The conceptual models and designs contained in a prototype, however, can usually be used in the final version. Precise specifications for the components of a prototype and clear documentation of its design are therefore critical for effective software prototyping.

Software prototyping has gained importance in recent years because new technologies have made computer-aided prototyping feasible. These technologies have reduced the time and cost involved in producing a prototype, thus widening the gap between a software prototype and the cost of the final software system, and increasing the potential leverage of prototyping.

The new technologies are based on reusable code, computer-aided design, and automatic generation of programs. The most powerful systems are designed for specific problem domains. Some problem domains for which computer-aided prototyping tools have been developed include business information processing (*see* ADMINISTRATIVE APPLICATIONS), user interfaces (*q.v.*), programming languages (*q.v.*), and real-time systems (*q.v.*).

Generators for business information systems provide graphical interfaces to databases to define database schemas, queries, and reports by graphically defining table layouts (*see* RELATIONAL DATABASES). There are many commercially available tools in this category.

Interface generation systems (Linton *et al.*, 1989) generate graphical user interfaces based on a set of predefined components such as windows, menus, scroll bars, and buttons. These components are placed and adapted interactively via a mouse and menu interface (*see* WINDOW ENVIRONMENTS).

Generators for language processors are mostly based on attribute grammars. These systems can generate various tools for computer languages based on a context-free grammar for the language, augmented with equations defining computed attributes for the nodes of the parse tree (*see* FORMAL LANGUAGES). This technology can be used to prototype tools for computer languages, including translators, interpreters, pretty-printers, type checkers, dataflow analyzers, and so forth. Applications span programming languages, specification languages, data definition languages for databases, hardware description languages (*q.v.*), and command languages for applications programs (*see* SCRIPTING LANGUAGES). Attribute grammar processors have been coupled to generators for syntax-directed editors (Reps and Teitelbaum, 1988) and program transformation systems (Abraido-Fandino, 1987).

Computer-aided prototyping of real-time systems (*q.v.*) is supported by the prototyping language PSDL (Luqi *et al.*, 1988) and the associated prototyping system CAPS. CAPS uses a software base of reusable components, a program generator, a static scheduler, and a dynamic scheduler to realize systems containing both functions with hard real-time constraints and non-time-critical functions. PSDL provides a simple representation of system decompositions using dataflow (*q.v.*) diagrams augmented with non-procedural control constraints and timing constraints (maximum response times, maximum execution times, minimum inter-stimulus periods, periods, and deadlines). The

language models both periodic and data-driven tasks, and both discrete (transaction-oriented) and continuous (sampled) data streams. The CAPS system provides automated tools for generating static schedules to guarantee hard real-time constraints as well as an execution support system that generates code for adapting, interconnecting, and controlling the execution of reusable software components.

In the future, prototyping will be integrated with final implementation. Progress in prototyping methods will hinge on automatic procedures for optimizing implementations. Prototypes will support refinement of the prototype into the final version by supplying additional information, and automatically transforming frequently used components to improve efficiency. Initially this will be done via optional implementation advice supplied by software engineers, in analogy to the *pragmas* in the programming language Ada (*q.v.*). In the longer term, prototyping systems will have reasoning capabilities and extensive knowledge bases which may include generic models of the problem domain, common goals of customers, common system structures, and generators producing specifications and code for classes of software components. Facilities for supporting formal verification of prototype decompositions are desirable to ensure that they are viable, especially if the subcomponents are to be built by different contractors.

Bibliography

1987. Abraido-Fandino, L. "An overview of REFINE 2.0," *Proceedings of the Second International Symposium on Knowledge Engineering*, Madrid, 8–10 April.

1988. Herndon, R., and Berzins, V. "The realizable benefits of a language prototyping language," *IEEE Trans. on Software Eng.*, **SE-14**, *6* (June), 803–809.

1988. Luqi, Berzins, V., and Yeh, R. "A prototyping language for real-time software," *IEEE Trans. on Software Eng.*, **14**, *10* (October), 1409–1423.

1988. Reps, T., and Teitelbaum, T. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. New York: Springer-Verlag.

1989. Linton, M., Vlissides, J., and Calder, P. "Composing user interfaces with interviews," *IEEE Computer*, **22**, *2* (February), 8–22.

1996. "Special issue on computer-aided prototyping," *Journal of Systems Integration*, **6**, *1–2*.

Valdis Berzins

SOFTWARE RELIABILITY

For articles on related subjects *see* DEBUGGING; PROGRAM SPECIFICATION; PROGRAM VERIFICATION; SOFTWARE MAINTENANCE; SOFTWARE METRICS; SOFTWARE MONITOR; SOFTWARE SAFETY; and SOFTWARE TESTING.

It is imperative to access the correctness of software for critical applications prior to actual use. Ideally we would like to verify formally that a program is correct. However, besides the practical difficulties encountered in applying current formal verification techniques to large programs, they cannot cope with the possibility of specification errors. An alternative approach is to use statistical methods to estimate the reliability of the software based on the outcome of program testing.

Software reliability is defined as the probability that a software fault that causes deviations from the required output by more than a specified tolerance, in a specified environment, does not occur during a specified exposure period. There are three distinct methods of estimating software reliability, namely, on the basis of its failure history, its behavior for a random sample of points taken from its input domain, or the number of seeded and actual faults detected by the test team where seeded faults are those that are deliberately inserted into the program at the start of the debugging phase.

Software Reliability Growth Models

Software reliability growth models attempt to predict the reliability of a program on the basis of its failure history. Failure history is defined to be the realization of a sequence of random variables $T_1, T_2, \ldots, T_n$, where T_i denotes the CPU time spent in testing the program after the faults causing the $(i-1)$th failure have been identified and removed until the ith failure is detected. These models can be further classified into fault-counting and nonfault-counting models depending on whether they express the reliability in terms of the number of faults remaining in the program or not.

Fault-counting models assume that the failure rates of the faults remaining in the program are independently identically distributed random variables and that the program failure rate is the sum of the individual failure rates. As an illustration, consider the General Poisson Model (GPM) which assumes that the failure rate, $r_j(t)$, after the faults causing the $(j-1)$th failure have been removed, is proportional to the number of faults remaining in the program and a power of the elapsed CPU time, i.e.

$$r_j(t) = \phi(N - M_j)\alpha t^{\alpha-1}$$

where α and ϕ are constants, N is the number of faults originally present in the program, and $M_j = \sum_{i=1}^{j} m_i$ where m_i is the number of faults removed following the ith failure. Hence, the reliability of the program after the jth failure is given by

$$R_j(t) = e^{-\phi(N-M_j)t^{\alpha}}.$$

Given $m_1, m_2, \ldots, m_n$ and $t_1, t_2, \ldots, t_n$, where t_j is the CPU time required to detect the jth failure after the