A DISCIPLINE FOR IMPROVING SOFTWARE PRODUCTIVITY

Robert M. Factor and William B. Smith

***Robert M. Factor** is director of the Analysis Systems Laboratory at AT&T Bell Laboratories, Middletown, New Jersey. Mr. Factor joined AT&T in 1971, and is responsible for the development of operations support systems. He has a B.S. in electrical engineering from the University of Denver, and both an M.S. in electrical engineering and Ph.D. in systems engineering from Case Western Reserve University. **William B. Smith** is executive director of the U.S. and International Operations Systems Division of Bell Laboratories at Middletown. His division develops operations systems for telecommunications companies here and overseas. Mr. Smith joined AT&T in 1962 after receiving a B.S. in electrical engineering from the University of Maryland. He subsequently received an M.S. in electrical*
(continued on page 9)

Software is the key to the functionality needed by today's telecommunications products and services. Traditionally, the creation of software has been a labor-intensive endeavor. Because of the tremendous growth in software needs during the past two decades, continuing emphasis has been given to finding ways to improve software development productivity. AT&T Bell Laboratories established a program in 1986 with the goal of tripling software development productivity within three years, i.e., improving it by a factor of three. This paper describes the program's framework and the technology being pursued to respond to this challenge.

Introduction

Since the opening of the modern telecommunications age in 1964 with the introduction of No. 1ESS™ switch and associated operations systems, software has played an expanding role in the achievement of new telecommunications functions both to meet customer needs through products and services and to operate and maintain the telecommunications network in a cost-effective manner. Today, significant AT&T revenues come from software; and, within a few years, software sales will begin to exceed hardware sales. Similar trends are evident with other products and services provided by the telecommunications industry.

Since its inception, software production has been a labor-intensive endeavor. At AT&T, about 40 percent of the total R&D community devotes most of its time to designing, coding, and testing software. An additional 20 percent supports the initial and final phases of the software product's life cycle: specifications or requirements and delivery. Combined, these efforts annually produce about 50 million lines of code in new products released to customers. Moreover, three-fourths of the people involved with software work in teams on large projects that require anywhere from 150,000 to four million lines of code.

With such tremendous growth in software needs during the past two decades, continuing attention has been focused on ways to make the software development process more productive. In the telecommunications business, software development costs are such a large percentage of sales that efficiency of product development will dominate a company's profitability and even its survivability. In the past, much of the effective academic work done to support software development productivity centered around "programming-in-the-small,"[1] tasks such as program editing, compiling, and debugging. For large software projects, however, productivity depends primarily on how "programming-in-the-large" tasks are performed and managed.[1] These tasks include specification, design, and test coordination and the supporting activities of software configuration management and version control. Building on what had previously been done at AT&T to improve productivity in all those software development tasks, the company in 1986 focused on a goal of improving productivity by a factor of three within three years. The factor was established not so much to measure software and productivity precisely as to pose a challenge to the R&D community. This issue of the *AT&T Technical Journal* describes several activities developed specifically in response to this challenge.

Obviously, introduction of a new technology cannot be applied uniformly to evolving and new projects. The degree to which a project is bound to existing technologies depends on its stage of development. Nonetheless, many of the new technologies can be applied to evolving projects and should produce significant productivity gains.

Specific projects are used as targets to validate new technologies and to obtain "industrial strength" — that is, high quality and robustness — in the tools that provide the technologies. Improved productivity will not only permit development of more new systems, but will also encourage replacing mature systems with ones that have lower software maintenance costs. The overall result will be better customer satisfaction, more revenues, and improved profitability.

Framework to Achieve Software Productivity Goals

The challenge to triple development productivity in three years demanded a program formulated to demonstrate results. Early on, we decided to adopt a rigorous system-engineering approach. Bell Laboratories had developed the discipline of systems engineering in the 1950s to conceive, implement, and deploy large products and complex services. The discipline has served the company well. Today, it represents about 15 percent of the effort within Bell Laboratories to define requirements for software-based projects within focused system-engineering groups. We wanted to use the same system-engineering focus to improve productivity in the software process itself.

Software System Engineering Process. The first step in the process was to model the effort needed to develop software for large projects (see Figure 1). Although software development within the AT&T R&D community includes all the traditional life-cycle stages,[2] activities in the different stages overlap in time, project, and personnel. Effort spent on any activity varies widely across projects. These variations from the traditional life cycle have evolved to accommodate the high degree of innovation and integration (with hardware and existing software) needed to build new telecommunication products. Despite the variations, we were able to map detailed models of the software development cycle into four stages: requirements, design, programming/unit test, and system test.

We did not directly address the maintenance stage of a product. Our goal was to improve productivity for the initial release. We reasoned that, if we reached our goal successfully, the same productivity gains would be carried into the product's maintenance phase.

In addition, we expected productivity increases to be accompanied by an increase in quality. Because software development is a labor-intensive endeavor, our expectation was based on the belief that technologies resulting in productivity improvements would reduce complexity in all stages of the development process, and that this reduction of process complexity would also improve quality.

Our early work required a definition of software metrics. We investigated alternative definitions of software metrics for productivity,[3] including function points. We concluded that the straightforward measure of "noncommentary lines of code per staff year," including reused code, was best for our initial purpose. (One outgrowth of

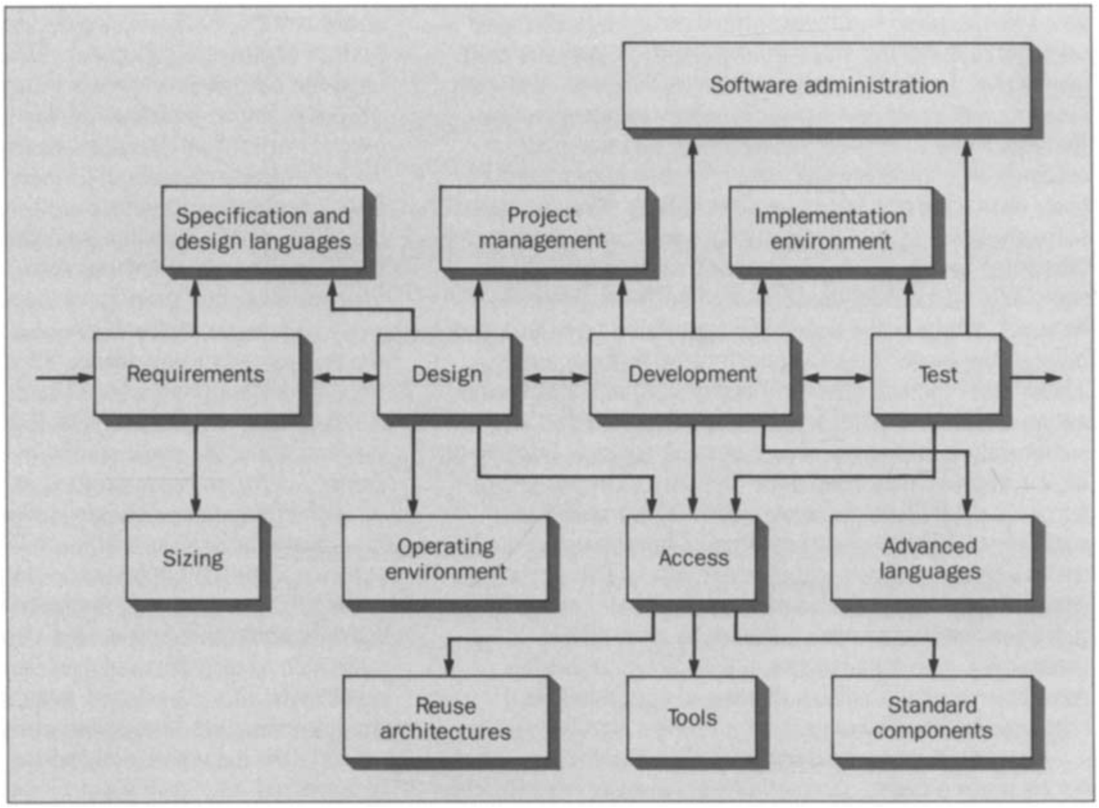

Figure 1. Software development environment.

this early work was a more rigorous program of measurement to develop a better base for comparison and estimation. The program is described in the paper by Lehder, Yu, and Smith in this issue.) This metric is easy to define and measure; no alternative metrics have demonstrated a clear superiority. In addition, because C programming language is a standard in the AT&T technical community, we have a very consistent measure.

Our work then turned to identifying the various components and important factors involved in developing large software projects. Surveys were taken within the AT&T R&D community to estimate how available human resources were distributed among the four stages of the development process. Although there was some spread by project, the mean was determined to be about 10 percent of effort in requirements, 20 percent in design, 50 percent in programming/unit test, and 20 percent in system testing. Experienced software managers and lead engineers then estimated how application of various technologies affects each step of the software development process. The value of these improvements was weighted by the percentage of effort spent on each stage to determine the most promising areas in which to apply either existing but not yet applied technology, or new technology.

Our results, for example, showed that productivity should double with increased component (building block) reuse. Early results from model projects have confirmed this estimate. Improvement by a factor of about 1.7 is possible with identifiable technology advances in software support tools (e.g., administration tools, automated test and analysis tools); a factor of 1.4 can be achieved by improving languages and code generators (also known as pattern or generation reuse). The combined application of these technologies, even discounting their mutual interactions, is expected to result in the desired productivity gain.

Technology Thrusts

Having concluded through engineering analysis that the desired productivity gains could be realized, we next analyzed areas of high technology (Figure 1) from which specific, high-priority technologies could be identified.

In selecting specific, high technologies, it is important to review technologies currently in use. The majority of software in AT&T is developed on UNIX® systems and uses C programming language. One advantage of the UNIX system environment is the availability of open interfaces that permit the introduction of many tools to support the software development process. For example, there are tools that profile a routine's real-time use, test its complexity and style, and check syntax and semantics against preset standards. The C language[4] itself, with its high level of portability and its extendibility via C++,[5] is a powerful system programming language used throughout the world. For programming-in-the-large tasks, MAKE, SCCS, and other change-management systems have existed on UNIX systems for a number of years.[6,7] MAKE is a UNIX system build program for software development; SCCS stands for "source code control system."

The challenge is to improve continually all these current facilities, both conceptually and at the performance level, and to come up with new facilities to integrate and automate many other tasks in the process. In evaluating new technology to answer that challenge, we considered potential payoff and the likelihood of success. As a result, we concluded that the following thrusts should be given highest priority in the near term:

- Product administration and configuration management
- Software reuse
- Languages
- Project management.

Product Administration and Configuration Management. One area of high-priority attention was improved software administration capability so that the state of the software could be put under stringent discipline throughout a product's life cycle.

A software product consists not only of the source code but also of associated source documents such as product installation scripts; *makefiles*; test scripts; modification requests (MRs); requirement, design, and user documents; and business and quality plans. Although each source component can have different versions and releases, all are related functionally (e.g., an MR may result in modifications to several files) and structurally (a particular product feature may have been implemented as a collection of sub-features). More than one person may be responsible for any component. In addition, as noted earlier, the entire product goes through several stages during its lifetime. These stages overlap in time, project and personnel. Disciplined administration of these components is crucial. Their interrelationships with other components, personnel, and life-cycle stages are similarly critical.

Concurrently, this administration must be automated as much as possible without intruding on the natural work habits of the people involved in the development. This required careful human engineering of the already evolving administration discipline and proper automation of its tasks. Two systems, SABLE and NMAKE, were central to this thrust. These systems, described by Cichinski and Fowler in this issue, were based on earlier UNIX system-based tools SCCS and MAKE.

Software Reuse. The biggest opportunity for productivity improvement was judged to be in the field of reuse. Reuse can be exercised at many levels of application. It has, in fact, been a mainstay of productivity improvement work for many years at AT&T.

Reuse can be obtained either by using existing components with little or no modification or by reusing templates. In the latter sense, generation is a form of reuse. If the domain of application is sufficiently understood and can be modeled accurately, then techniques exist to create systems with program generators. New approaches to producing these generators from the application descriptions have been formulated and applied in certain projects. These approaches are described in the paper by Cleaveland and Kintala in this issue.

For telecommunications applications, however, the greatest productivity payoffs result when large existing

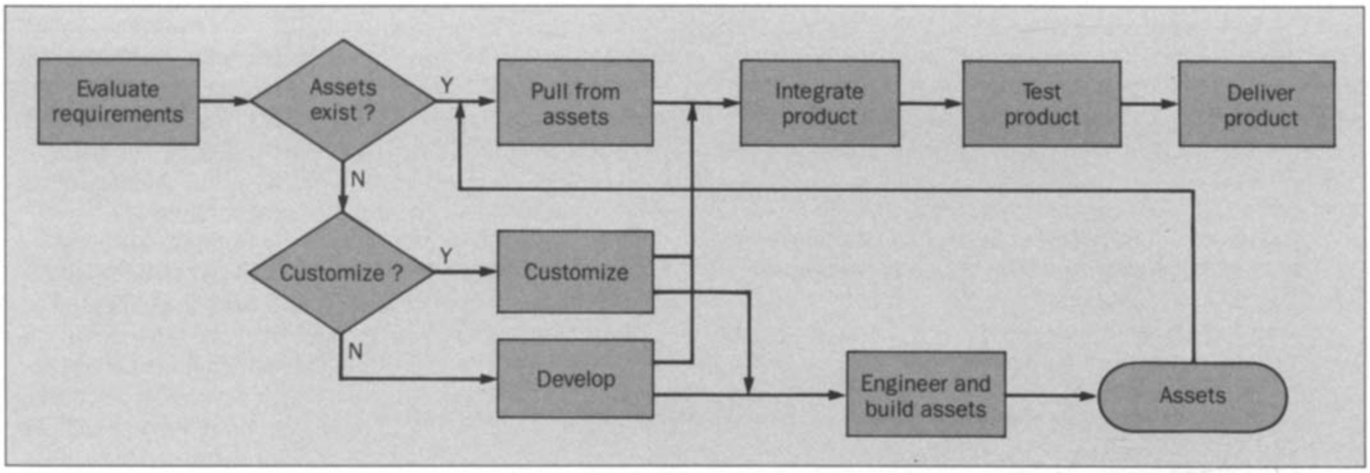

Figure 2. Process model for asset reuse.

components are reused. One level of component reuse is to build successive software generic releases on previous releases of the same application. (We apply the term "generic" to a series of software products that serve a broad functional area to recognize this level of reuse.) A second level of reuse is to apply standard system components and services available with the operating system embedded in the product; one of the strengths of the UNIX operating system is the set of components and services available with it. The reuse of application components borrowed from a close, cooperative association with other projects represents a third level of reuse.

The reuse area of greatest importance to our current software productivity effort is the utilization of components *designed for reuse* as part of an engineering plan. These components, truly envisioned as assets, may be relatively small (such as string manipulation routines), of moderate size (such as error handling packages), or rather large (such as recovery or system reconfiguration subsystems).

Figure 2 shows a version of an asset-reuse model that describes the essence of a design process based on engineered reuse. The first step in the process is to utilize reusable assets from a pool to obtain a desired solution. If all assets already exist in the pool, then a system development effort would consist of:

- Pulling the assets together to provide an integrated software product that includes user and administrative documentation
- Testing the product to assure that it has met requirements
- Delivering the product to the customer.

Currently, reusable assets that meet the needs of all the complex telecommunications products we build are rare. However, certain classes of telecommunications products can benefit substantially from existing software assets.

In addition, a process has been created to obtain engineered assets both by cataloging existing assets and choosing those that fit the "engineered" environment (including system performance requirements) and by contracting internally for developments to provide new assets consistent with their developments and the engineering specification for reusable assets. These assets are then placed into the asset pool for use on future developments. The paper by Anderson, Beck, and Buonanno in this issue describes the basis leading to large-scale component reuse through modular components.

Languages. A key tool in any software development activity is, of course, the programming language used to write the software's programs. In the past, AT&T had moved from using assembly languages to using high-level languages, such as C, to build its telecommunications products. The use of high-level languages reduces the complexity associated with the programming process. It does so by moving away from the need to base programs on the intrinsic nature of computers and toward programming based on the nature of the problem.

Systems currently being built, and telecommunications products envisioned for the future, could be 10 times more complex than earlier systems in control and data-flow structures, interprocedure interactions, and volume and efficiency requirements. Yet, to improve productivity levels, we need to reduce the complexity of the programming process. To do so, we want to use more expressive power in the general-purpose languages, and more languages designed for special-purpose applications.

C++, an extension of C, provides support for data abstraction and object-oriented programming paradigms. The ability of C++ to encapsulate related application concepts into a single structure, called a *class*, opens the way to designing systems that promote reuse in the programming process and complements the component reuse described in the preceding section. Its *inheritance* mechanism reduces the effort of handling the complexity mentioned above. At the same time, C++ is compatible with existing C software and expertise, both important AT&T assets. This compatibility is essential for its rapid and graceful introduction into a wide variety of AT&T projects. The paper by Coplien, Dewhurst, and Koenig in this issue describes C++ in this context.

Application languages provide a second way to improve productivity. It is more natural and, therefore, more productive for the application builder to specify and program in a language that has the application concepts built into it. AWK[8] is an excellent example of an application language for writing programs to manipulate data stored on sequential files on the UNIX system. Languages with finite-state-machine concepts built into them will be natural for the call-processing applications in switching systems. These languages can also help developers specify and build prototypes rapidly. The paper by Brown et al. in this issue explains the concepts in greater detail.

Project Management. Providing a methodology (procedures and documentation) to manage the development process is a key part of the productivity improvement program. An important step is the definition and documentation of a methodology that could be applied to targeted projects. In AT&T, the software development methodology is featured in the Software Project Management Workshop, a training course described in an earlier issue of the *AT&T Technical Journal.*[9] During this one-week workshop, four principles of effective management—planning, organizing, monitoring, and controlling—are taught through exercises and lectures. The end result of the workshop is an actual project plan generated by participants. The workshop's curriculum was built on AT&T's experiences with both successful and unsuccessful projects, as well as from published work.[10,11] Methodologies must, of course, be backed by associated tools to permit ease of implementation. Product administration and configuration management (described by Cichiński and Fowler's paper in this issue) and the software estimation technology (see the paper by Lehder, Yu, and Smith, also in this issue) are examples of tools developed directly to support the methodology's basic discipline.

Future Directions. Much of the technology used to improve software development productivity was initiated in research programs during the past decade. Thus, as part of our ongoing productivity program, we want to support continuing research efforts and find ways to transfer the resulting technology rapidly into development projects. Considerable research is now being done on methods such as increasing computer aids in the front end of the design process and generation of software; program-flow-structure visualization; integrated version, build, and test environments; and specification languages. A number of distributed workstation trials are under way that experiment with approaches for improved implementation and execution environments. These and other areas of investigation that show promise of future gains are described in the paper by Belanger, Bergland, and Wish in this issue.

Relationship of Technologies to Productivity Opportunities

The state-of-the-art technologies and their application, reported in this issue, were chosen for their anticipated payoff in improving our software development process. Software projects throughout AT&T abound with evidence of immediate productivity improvements that result directly from application of these technologies. For example, SABLE is used in more than 180 projects. When SABLE is used, new projects come on board within a day because the system automatically sets up the entire mechanism to administer the project and manage changes to it. Similarly, NMAKE routinely decreases the time interval by factors of five or more when compared to technology used in MAKE, NMAKE's predecessor. Reusable components have also helped reduce the time intervals between new products.

It is obviously difficult to measure precisely productivity improvements attributable to these technologies. This is because of the interrelationships among aspects of software development and the idiosyncrasies in different project environments. Moreover, as noted earlier, new technology cannot be introduced uniformly in both evolving and new projects because the degree of commitment to existing technologies is a function of a project's stage of development.

Another approach to determining payoff is similar to the one advocated recently by Boehm.[12] His conclusions are stated as a set of six opportunities to drive down software development costs:

- Get the best from people
- Make steps more efficient
- Eliminate steps
- Eliminate rework
- Build simpler products
- Reuse components.

Our technology thrusts map into these opportunities. Some of these mappings are straightforward and obvious; others are not.

For example, we believe that effective estimation techniques will improve productivity in two ways. First, we get the best from people if we can accurately estimate the effort and schedules for a product. If schedules are too tight, inefficiencies occur and quality drops. Second, realistic estimates have a tendency to drive projects to build simpler products, or to increase component reuse because of the recognition that complex customized developments involve higher costs and risks.

The focus is on the most important issues involved in providing the functionality required of the development. It is significant to note that we use a process discipline that forms the framework for development activities, not a methodology that gives step-by-step instructions. We encourage variances in the discipline to allow for local differences and for the discipline's growth.

Productivity and Quality. No discussion about productivity would be complete without considering the interaction between quality and productivity. Although the papers in this issue focus on technologies that improve productivity directly, quality has an important and direct relationship with productivity. Improvements in quality reduce rework, and reduced rework means increased productivity. AT&T's aggressive quality program[13] complements the productivity work described here.

Quality in software products is often presented in the context of the system's error-free behavior in user environments.[2,3,11] For AT&T to be a leader in the production of high-quality software, the software must not only be free of "bugs," but must also satisfy many other customer expectations. It must, as well, include attributes such as elegance in design, reduced complexity, transparency of the design in the implementation through use of high-level languages, and smooth integration of the information generated during the various stages of the system that map onto the ability to evolve rapidly to meet user needs. The tools that represent the technologies described in this issue help in these additional aspects of quality. For example, C++ reduces the complexity of the design paradigm by letting a developer work with high-level conceptual objects that encapsulate into a single structure both the data and the operations on data. Reusable modules simplify software development by providing plug-compatible components, thereby allowing the system developer to concentrate on the essential elements of the new functionality that is to be added. Reduced complexity directly increases both productivity and quality.

Technology Transfer. Finally, to obtain desired increases in productivity and quality, the technologies must evolve. They must be embedded in the industrial-strength tools used both to build the products we provide to our customers and in the products themselves. Obtaining industrial strength in the tools requires a transfer of the technologies from research to general use through a controlled process.[14] At AT&T, the organization whose charter is to improve the R&D community's productivity through technology transfer recognized the increasing importance of its task. The organization has increased its efforts to make protoypes of these technologies, then test them in a manner identical to all other AT&T products. It recognizes that technologies that improve productivity must also improve quality. The tools and building blocks that embody the technology must, therefore, also be of extremely high quality.

Conclusion

Leadership in telecommunications products and services depends not only on world-class hardware technology, but also increasingly on customer satisfaction based on functionality, quality, timeliness, and cost of software products. During the past 20 years, much progress has been made to convert the art of software development into an engineering discipline that better serves customer needs. The papers in this issue represent some of the steps taken within AT&T. The reported improvements, however, are just part of a journey that will not end as long as customers continue to look for even more effective ways to meet their telecommunications needs.

References

1. F. DeRemer and H. H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, June 1976, pp. 80-86.
2. B. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
3. C. Jones, *Programming Productivity*, McGraw-Hill, New York, 1986.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
5. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.
6. UNIX Time-Sharing System, *Bell System Technical Journal*, July/August 1978, Vol. 57, No. 6, Part 2.
7. *UNIX System V User Reference Manual, Release 3.2*, AT&T, June 1988.
8. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, Massachusetts, 1988.
9. L. B. Robertson and G. A. Secor, "Effective Management of Software Development," *AT&T Technical Journal*, Vol. 65, No. 2, March/April 1986, pp. 94-101.
10. P. B. Crosby, *Quality is Free*, McGraw-Hill, New York, 1979.
11. T. Demarco, *Controlling Software Projects*, Yourdin Press, New York, 1982.
12. B. W. Boehm, "Improving Software Productivity," *Computer*, Vol. 20, No. 9, September 1987, pp. 43-57.
13. Quality: Theory and Practice, *AT&T Technical Journal*, Vol. 65, No. 2, March/April 1986.
14. D. G. Belanger, "Technology Transfer: A Supplier View," *IEEE Second Workshop on Software Technology Transfer*, June 1987.

Biographies (continued)

engineering from Princeton University and a Ph.D. in con puter science from the University of Pennsylvania.

(Manuscript received May 24, 1988)