Improving Software Productivity

Barry W. Boehm, TRW

Computer hardware productivity continues to increase by leaps and bounds, while software productivity seems to be barely holding its own. Central processing units, random access memories, and mass memories improve their price-performance ratios by orders of magnitude per decade, while software projects continue to grind out production-engineered code at the same old rate of one to two delivered lines of code per man-hour.

Yet, if software is judged by the same standards as hardware, its productivity looks pretty good. One can produce a million copies of Lotus 1-2-3 at least as cheaply as a million copies of the Intel 286. Database management systems that cost $5 million 20 years ago can now be purchased for $99.95.

The commodity for which productivity has been slow to increase is custom software. Clearly, if you want to improve your organization's software price-performance, one major principle is "Don't build custom software where mass-produced software will satisfy your needs." However, even with custom software, a great deal is known about how to improve its productivity, and even increasing productivity by a factor of 2 will make a significant difference for most organizations.

This article discusses avenues of improving productivity for both custom and mass-produced software. Its main sections cover the following topics:

- The importance of improving software productivity: some national, international, and organizational trends indicating the significance of improving software productivity.
- Measuring software productivity: some of the pitfalls and paradoxes in defining and measuring software productivity and how best to deal with them.
- Analyzing software productivity: identifying factors that have a strong productivity influence and those that have relatively little influence, using such concepts as software productivity ranges, the software value chain, and the software productivity opportunity tree.
- Improving software productivity: using the opportunity tree as a framework for describing specific productivity improvement steps and their potential payoffs.
- Software productivity trends and conclusions.

The importance of improving software productivity

The major motivation for improving software productivity is that software costs are large and growing larger. Thus, any percentage savings will be large and growing larger as well Figure 1 shows recent and projected software cost trends in the United States and worldwide. In 1985, software costs totaled roughly $11 billion in the US Department of Defense, $70 billion in the United States overall, and $140 billion worldwide. If present software cost growth rates of approximately 12 percent per year continue, the 1995 figures will be $36 billion for the DoD, $225 billion for the United States, and $450 billion worldwide. Thus, even a 20 percent improvement in software productivity would be worth $45 billion in 1995 for the United States and $90 billion worldwide. Gains of such magnitude are clearly worth a serious effort to achieve.

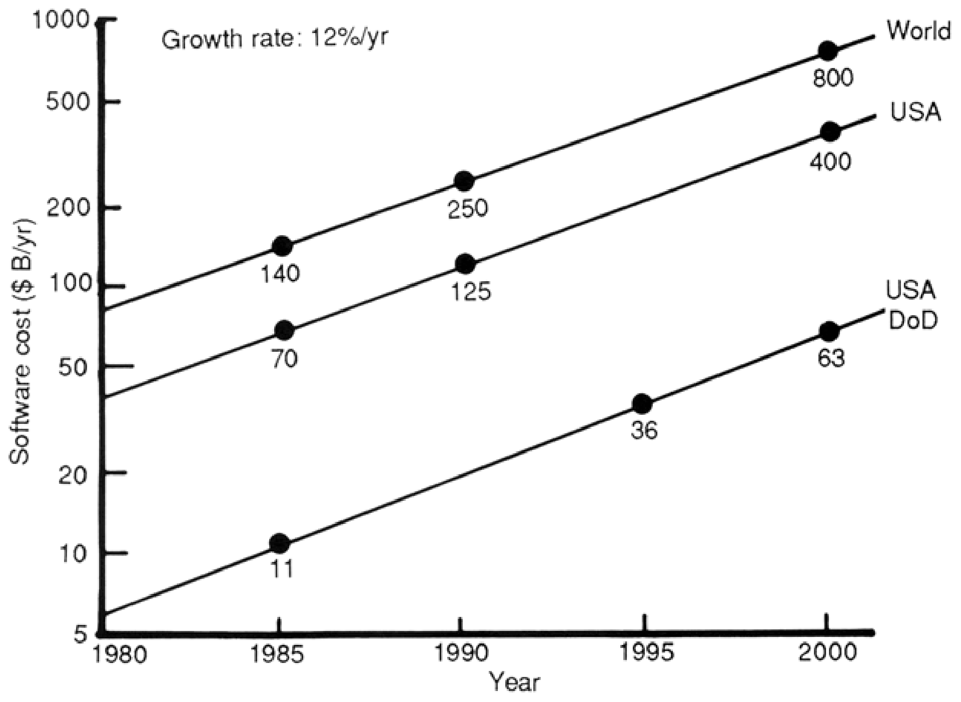


Figure 1. Software cost trends.

Software costs are increasing not because people are becoming less productive but because of the continuing increase in demand for software, Figure

2, based on Boehm[1] and a recent TRW-NASA Space Station software study, shows the growth in software demand across five generations of the U.S. manned space flight program. from about 1,500,000 object instructions to support Project Mercury in 1962–63 to about 80,000,000 object instructions to support the Space Station in the early 1990's.

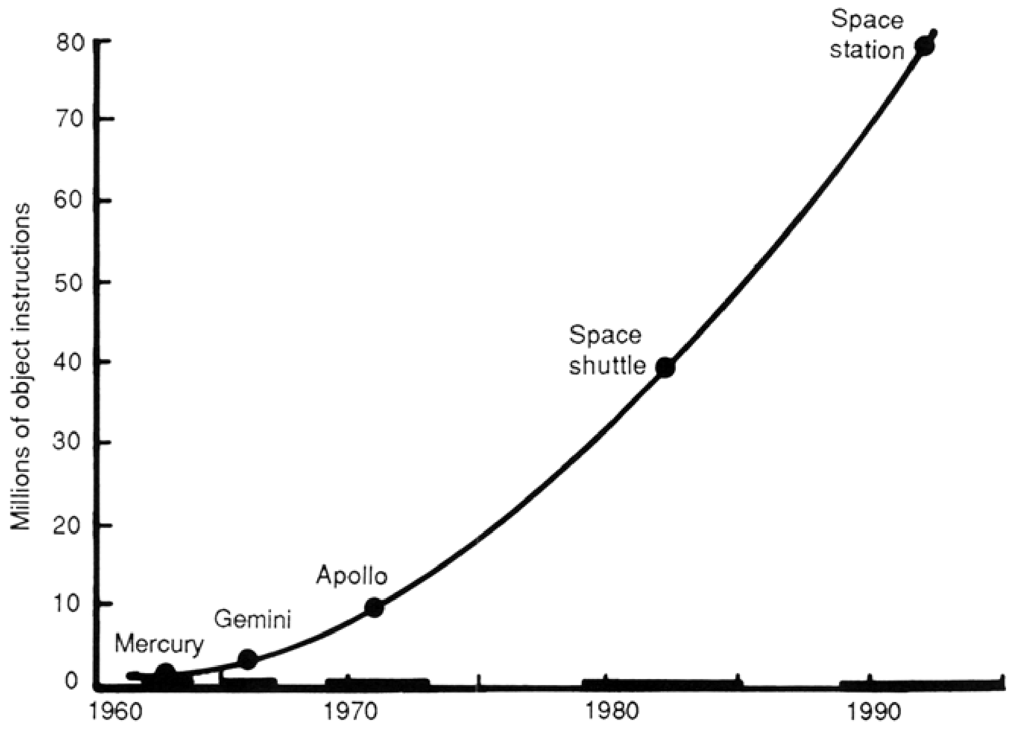


Figure 2. Growth in software demand: US manned spaceflight program.

The reasons for this increased demand are basically the same ones encountered by other sectors of the economy as they attempt to increase productivity via automation. The major component of growth in the Space Shuttle software has been the checkout and launch support area, in which NASA automated many functions to reduce the number of people needed to support each launch—as many as 20,000 in previous manned spaceflight operations. The result has been a significant reduction in required launch support personnel but a significant increase in the required amount of software.

Many organizations have software demand growth curves similar to Figure 2. A large number of organizations simply cannot handle their increased demand within their available personnel and budget constraints, and they are faced with long backlogs of unimplemented information processing systems and software improvements. For example, the U.S. Air Force Standard Information Systems Center has identified a four-year backlog of unstarted projects representing user-validated software needs. This type of backlog serves as a major inhibitor of a software user organization's overall productivity, competitive-ness, and morale. Thus, besides cost savings, another major motivation for improving software productivity is to break up these software logjams.

Measuring software productivity

The best definition of the productivity of a process is

$$\text{Productivity} = \frac{\text{Outputs produced by the process}}{\text{Inputs consumed by the process}}$$

Thus, we can improve the productivity of the software process by increasing its outputs, decreasing its inputs, or both. However, this means that we need to provide meaningful definitions of the inputs and outputs of the software process.

Defining inputs. For the software process, providing a meaningful definition of *inputs* is a nontrivial but generally workable problem. Inputs to the software process generally comprise labor, computers, supplies, and other support facilities and equipment. However, one has to be careful which of various classes of items are to be counted as inputs. For example:

- Phases (just software development, or should we include system engineering, soft-ware requirements analysis, installation, or postdevelopment support?)
- Activities (to include documentation, project management, facilities management, conversion, training, database administration?)
- Personnel (to include secretaries, computer operators, business managers, contract administrators, line management?)
- Resources (to include facilities, equipment, communications, current versus future dollar payments?)

An organization can usually reach an agreement on which of the above are meaningful as inputs in their organizational context. Frequently, one can use present-value dollars as a uniform scale for various classes of resources.

Defining outputs. The big problem in defining software productivity is defining *outputs*. Here we find a paradox. Most sources say that defining delivered source instructions (DSI) or lines of code as the output of the software process is totally inadequate, and they argue that there are a number of deficiencies in using DSI. However, most organizations doing practical productivity measurement still use DS1 as their primary metric.

DSI does have the following deficiencies as a software productivity metric:

(1) It is too low-level for some purposes, particularly for software cost estimation, where it is often difficult to estimate DSI in advance.

(2) It is too high-level for some purposes because complex instructions or complex combinations of instructions receive the same weight as a sequence of simple assignment statements.

(3) It is not a uniform metric; lines of machine-oriented language (MOL), higher-order language (HOL), and very high level language (VHLL) are given the same weight. For example, completing an application in one man-month and 100 lines of VHLL (100 DSI/MM) should not be considered less productive than doing the same application in two man-months and 500 lines of HOL (250 DSI/MM).

(4) It is hard to define well, particularly in determining whether to count comments, nonexecutable lines of code, reused code, or a "line" as a card image, carriage return, or semicolon. For example, putting a compact Ada program through a pretty printer will frequently triple its number of card images.

(5) It is not necessarily well correlated with value added, in that motivating people to improve productivity in terms of DSI may tempt them to develop a lot of useless lines of code.

(6) It does not reflect any consideration of software quality; "improving productivity" may tempt people to produce faster but sloppier code.

A number of alternatives to DSI have been advanced:

- "Software science" or program information-content metrics
- Design complexity metrics
- Program-external metrics, such as number of inputs, outputs, inquiries, files interfaces, or function points, or a linear combination of those five quantities[2]
- Work transaction metrics

Comparing the effectiveness of these productivity metrics to a DSI metric, the following conclusions can be advanced: each has advantages over DSI in some situations, each has more difficulties than DSI in some situations, and each has equivalent difficulties to DSI in relating software achievement units to measures of the software's value added to the user organization.

As an example, let us consider function points, which are defined as

$$\text{FPs} = 4 \text{ x \#Inputs} + 5 \text{ x \#Outputs} + 4 \text{ x \#Inquiries} + 10 \text{ x \#Masterfiles} + 7 \text{ x \#Interfaces},$$

where #Inputs means "number of inputs to the program," and so on for the other terms.

Function points offer some strong advantages in addressing problems 1 (too low-level) and 3 (nonuniformity) above. One generally has a better early idea of the number of program inputs, outputs, and so on, and the delivered software functionality has the same numeric measure whether the application is implemented in an MOL, HOL, or VHLL. However, function points do not provide any advantage in addressing problems 5 and 6 (value added and quality considerations), and they have more difficulties than DSI with respect to problems 2 and 4 (too high-level and imprecise definition). The software functionality required to transform an input into an output may be very trivial or very extensive. And we still lack a set of well-rationalized, unexceptionable standard definitions for number of inputs, number of outputs, and other terms that are invariant across designers of the same application. For example, some experiments have shown an order-of-magnitude variation in estimating the number of inputs to an application.

However, function points have been successfully applied in some limited, generally uniform domains such as small-to-medium-sized business applications programs. A number of activities are also under way to provide more standard counting rules and to extend the metric to better cover other software application domains.

Thus, no alternative metrics have demonstrated a clear superiority to DSI. And DSI has several advantages that induce organizations to continue to use DSI as their primary software productivity output metric:

- The DSI metric is relatively easy to define and discuss unambiguously.
- It is easy to measure.
- It is conceptually familiar to software developers.
- It is linked to most familiar cost estimation models and rules of thumb for productivity estimation.
- It provides continuity from many organizations' existing database of project productivity information.

Software productivity-quality interactions. As discussed above, we want to define *productivity* in a way that does not compromise a project's concern with software quality. The interactions between software cost and the various software qualities (reliability, ease of use. ease of modification. portability, efficiency, etc.) are quite complex, as are the interactions between the various qualities them-selves. Overall, though, there are two primary situations that create significant interactions between software costs and qualities:

(1) A project can reduce software development costs at the expense of quality but only in ways that increase operational and life-cycle costs.
(2) A project can simultaneously reduce software costs and improve software quality by intelligent and cost-effective use of modern software techniques.

One example of situation 1 was provided by a software project experiment in which several teams were asked to develop a program to perform the same function, but each team was asked to optimize a different objective. Almost uniformly, each team finished first on the objective they were asked to optimize, and fell behind on the other objectives. In particular, the team asked to minimize effort finished with the smallest effort to complete the program, but also finished last in program clarity, second to last in program size and required storage, and third to last in output clarity.

Another example is provided by the COCOMO database of 63 development projects and 24 evolution or maintenance projects[1]. This analysis showed that if the effects of other factors such as personnel, use of tools, and modern programming practices were held constant, then the cost to develop reliability-critical software was almost twice the cost of developing minimally reliable software. However, the trend was reversed in the maintenance projects; low-reliability software required considerably more budget to maintain than high-reliability software. Thus, there is a "value of quality" that makes it generally undesirable in the long run to reduce development cost at the expense of quality.

Certainly, though, if we want better software quality at a reasonable cost, we are not going to hold constant our use of tools, modern programming practices, and better people. This leads to situation 2, in which many organizations have been able to achieve simultaneous improvements in both software quality and productivity. For example, the extensive Guide, Inc., survey of about 800 user installations found that the four most strongly experienced effects of using modem programming practices were code quality, early error detection, programmer productivity, and maintenance time or cost. Also, the COCOMO life-cycle data analysis indicated that the use of modern programming practices had a strong positive impact on development productivity but an even stronger positive impact on maintenance productivity.

However, getting the right mix of the various qualities (reliability, efficiency, ease of use, ease of change, etc.) can be a very complex job. Several studies have explored these qualities and their interactions. Also, several new approaches have had some success in providing methods for reconciling and managing multiple quality objectives, such as Gilb's design by objectives and the Goals approach (Boehm,[1] Chapter 3). For pointers to additional information on these and other topics covered in this article, see the "Further Reading" section.

Metrics: The current bottom line. The current bottom line for most organizations is that delivered source instructions per project man-month (DSI/MM) is a more practical productivity metric than the currently available alternatives. To use DSI/MM effectively, though, it is important to establish a number of measurement standards and interpretation guidelines, including

- Objective, well-understood counting rules defining which project-related man-months are included in MM;
- Objective, well-understood counting rules for source instructions
- A definition of *delivered* in terms of compliance with a set of software quality standards;
- Definition and tracking of the language level and extent of reuse of source instructions, along with interpretation guidelines encouraging the use of VHLLs, HOLs, and reused software.

Examples of such definitions are given by Boehm[1] and by Jones.[2]

In addition, because new metrics such as function points have been successful in some areas, many organizations are also experimenting with their use, refinement, and extension to other areas.

Analyzing software productivity

We can consider two primary ways of analyzing software productivity:

(1) The "black-box" or influence-function approach, which performs comparative analyses on the overall results of a number of entire software projects, and which tries to characterize the overall effect on software productivity of such factors as team objectives, methodological approach, hardware constraints, turnaround time, or personnel experience and capability.

(2) The "glass-box" or cost-distribution approach, which analyzes one or more soft-ware projects to compare their internal distribution between such costs

as labor and capital, code and documentation, development and maintenance, and other cost distributions by phase or activity.

Here, we will concentrate on two representative approaches: the black-box productivity range and the glass-box value chain.

Software productivity ranges. Most software cost estimation models incorporate a number of software cost driver factors:
attributes of a software project or product that affect the project's productivity in (appropriately defined) DSI/MM. A significant feature of some of these models is the productivity range for a software cost driver: the relative multiplicative amount by which that cost driver can influence the software project cost estimated by the model. An example of a set of recently updated productivity ranges for the COCOMO models is shown in Figure 3.

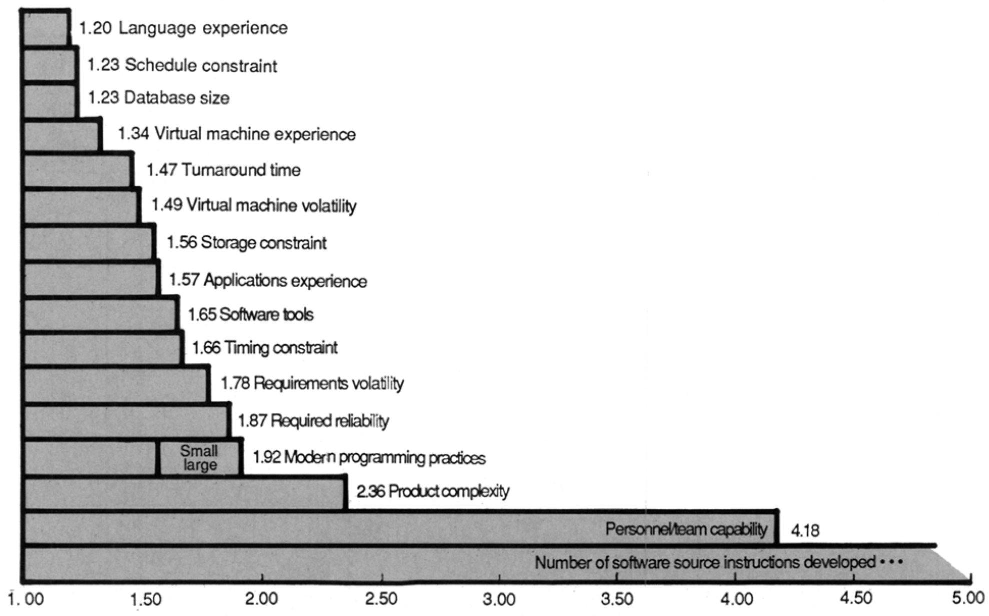


Figure 3. Cocomo software life-cycle productivity ranges, 1985.

These productivity ranges show the relative leverage of each factor on one's ability to reduce the amount of effort required to develop a software product. For example, assuming all the other factors are held constant, developing a software product in an unfamiliar programming language will typically require about 20 percent more man-months than using a very familiar language. Similarly, developing a product with a mediocre (15th-percentile) team of people will typically require over four times as many man-months as with a 90th-percentile team of people. The open-ended bar at the bottom of Figure 3 indicates that the number of man-months required to develop a software product increases without bound as one increases the number of instructions developed.

Some initial top-level implications of the productivity ranges are summarized as follows; more detailed implications will be discussed in the “Improving Software Productivity” section later in this article.

• *Number of source instructions.* The most significant influence on software costs is the number of source instructions one chooses to program. This leads to cost reduction strategies involving the use of fourth-generation languages or reusable components to reduce the number of source instructions developed, the use of prototyping and other requirements analysis techniques to ensure that unnecessary functions are not developed, and the use of already developed software products.

• *Management of people.* The next most significant influence by far is that of the selection, motivation, and management of the people involved in the software process. In particular, employing the best people possible is usually a bargain, because the productivity range for people usually is much wider than the range of people’s salaries.

• *Fixed features of the product.* Some of the factors, such as product complexity, required reliability, and database size, are largely fixed features of the software product and not management controllables. Even here, though, appreciable sayings can be achieved by reducing unnecessary complexity and by focusing on appropriate life-cycle cost-reliability trade-offs as discussed in the preceding section.

• *Other, management-controllable factors.* The other cost driver factors are generally management controllables: requirements volatility, hardware speed and storage constraints, use of software tools and modem programming practices, and so on can be directly factored into a productivity improvement effort.

Some primary productivity improvement strategies involving these cost driver variables are described later. See Boehm[1], Chapter 33, for a discussion of each cost driver and Boehm et al.[3] for an example of their successful application to an integrated soft-ware productivity improvement program.

The software product value chain. The value chain, developed by Porter and his associates at the Harvard Business School[4], is a useful method of understanding and controlling the costs involved in a wide variety of organizational enterprises. It identifies a canonical set of cost sources or value activities, representing the basic activities an organization can choose from to create added value for its products. Figure 4 shows a value chain for software development representative of experience at TRW. Definitions and explanations of the component value activities are given by Porter[4].

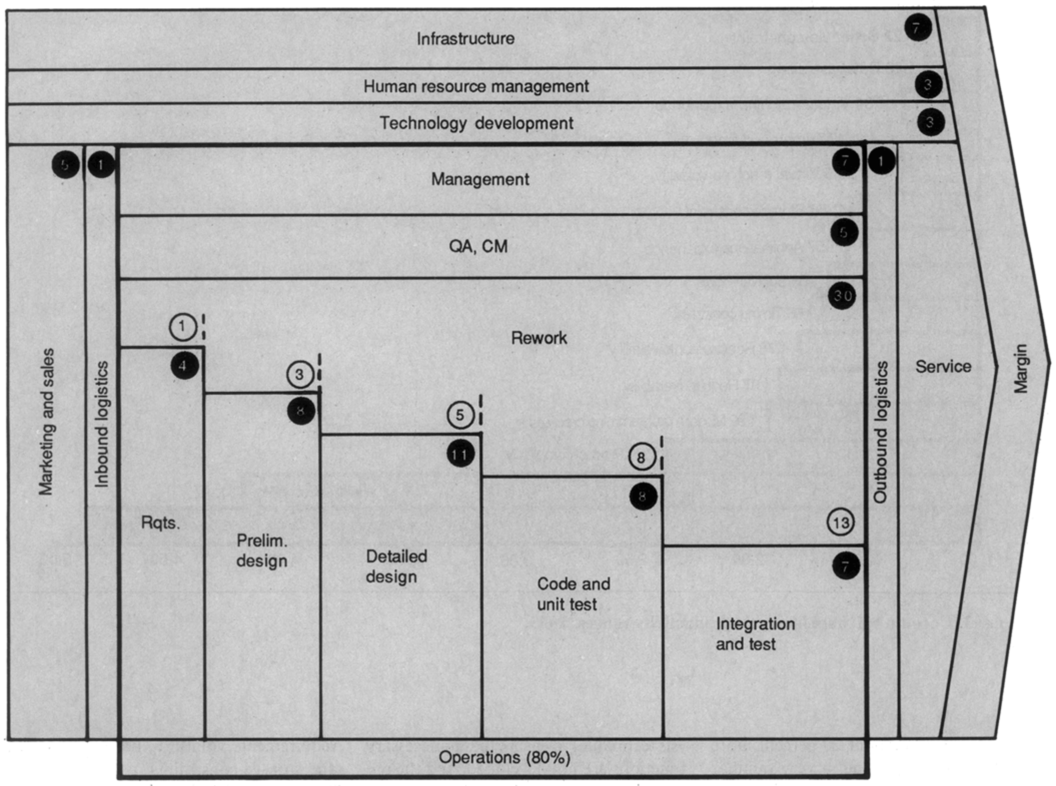


Figure 4. Typical software development value chain.

For software, the largest single value chain element is Operations, which covers activities associated with transforming inputs into the final product form and typically involves roughly four-fifths of the total development outlay. In such a case, the value chain analysis involves breaking up a large component into constituent activities. Figure 4 shows such a breakup of Operations into management (7 percent), quality assurance and con-figuration management (5 percent), and the distribution of technical effort among the various development phases. This phase breakdown also covers the cost sources due to rework. Thus, for example, of the 20 percent overall cost of the technical effort during the integration and test phase; 13 percent is devoted to activities required to rework deficiencies in or reorientations of the requirements, design, code, or documentation; and the other 7 percent represents the amount of effort required to run tests, perform integration functions, and complete documentation even if no problems were detected in the process.

For simplicity, the service and margin components of the value chain have not been assigned numerical values. "Margin" basically represents profits; "service" represents postdevelopment software support activities, often called "maintenance" but more properly called "evolution. " Evolution costs are typically 70 percent of software life-cycle costs, but since some initial analyses have indicated that the detailed value-chain distribution for evolution costs is not markedly different from the distribution of development costs in Figure 4, we will use Figure 4 to represent the distribution of software life-cycle costs.

The primary implication of the software development value chain is that the Operations component is the key to significant improvements. Not only is it the major source of software costs, but also most of the remaining components such as human resources will scale down in a manner roughly proportional to the scaling down of Operations cost.

Another major characteristic of the value chain is that virtually all of the components are still highly labor intensive. Thus, there are significant opportunities for providing automated aids to make these activities more efficient and capital intensive. Further, it implies that human resource and management activities aimed at *getting the best from peopie* have much higher leverage than their 3 percent and 7 percent investment levels indicate.

The breakdown of the Operations component indicates that the leading strategies for cost savings in software development involve

- *making individual steps more efficient* via such capabilities as automated aids to software design analysis or testing;
- *eliminating steps* via such capabilities as automatic programming or automatic quality assurance;
- *eliminating rework* via early error detection or via such capabilities as rapid prototyping to avoid later requirements rework.

In addition, further major cost savings can be achieved by reducing the total number of elementary operations steps by developing products requiring the creation of fewer lines of code. This has the effect of reducing the overall size of the value chain itself. This source of savings breaks down into two main options:

- *Building simpler products* by applying more insight to front-end activities such as prototyping or risk management;
- *Reusing software components* via such capabilities as fourth-generation languages or component libraries.

The software productivity improvement opportunity tree. This breakdown of the major sources of software cost savings leads to the software productivity improvement opportunity tree shown in Figure 5. This hierarchical breakdown helps us to understand how to fit the various attractive productivity options into an overall integrated software productivity improvement strategy. The next section will discuss each of these major options in turn.

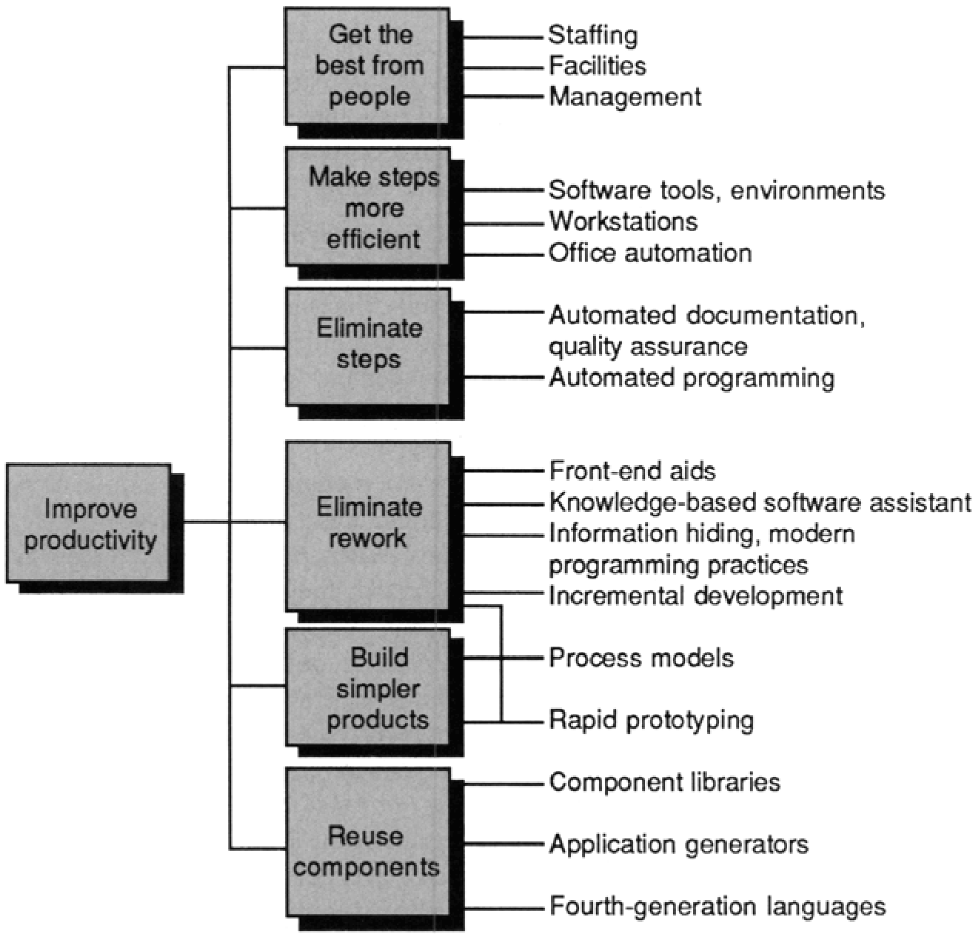


Figure 5. Software productivity improvement opportunity tree.

Improving software productivity

Following the organization of the software productivity opportunity tree, we will cover the following primary options for improving software productivity: (1) getting the best from people, (2) making steps more efficient, (3) eliminating steps, (4) eliminating rework, (5) building simpler products, and (6) reusing components.

Getting the best from people. As indicated in the opportunity tree, there are three primary options available for getting the best from people: staffing, facilities, and management.

Staffing. The productivity ranges in Figure 3 show a factor of 4.18 in productivity difference due to personnel/team capability and a combined factor of 2.52 for relative experience with the applications area, computer system or virtual machine, and programming language. Similar ranges have been determined by other studies such as the IBM productivity analysis (see Walston and Felix[5]).

Thus, if you want to increase your project's or organization's software productivity, one of the biggest leverage actions you have at your disposal is to get the best people working for your project or organization and the mediocre people working for someone else. It is worth making a significant effort to get this to happen. But it is remarkable how frequently managers are passive about key staffing decisions and how frequently they go in the opposite direction, saying things like

> "I can't afford those high-salary people."
> "1 can't take a risk on somebody so expensive."
> "I can't hire your superstar until your project gets its funds, even though that's only a month away."
> "Joe has all these unassigned people charging to standby, and I have to help him out."
> "I can't wait. I need somebody to show some progress on this task by next week."

Sometimes, the latter two situations require you to respond, but you can generally make it a temporary rather than a permanent commitment.

The other, equally important, side of the staffing coin involves committing yourself to phase out misfits. No matter how carefully you select the members of your software team, inevitably you will find some people who do not contribute anywhere near their fair share to the team's objectives, even after several attempts to find on prepare a suitable role for them on the team. In such a situation, you will be tempted to postpone dealing with the problem, to profess not to notice it, to smooth it over with words, or to ask the other team members to do extra tasks. This may be the easy way out in the short run, but invariably it produces unhealthy results in the long run.

Phasing people out isn't easy, but if you devote enough time, thought, and sympathy to the problem, you can often create a situation in which the phaseout becomes a positive rather than a negative experience, and the person concerned finds a new line of work that suits him or her much better than a group-oriented software project. If this doesn't work, and you are left with a definite misfit, don't back away from the problem. Get rid of the misfit as quickly as possible.

Facilities. Given that software development and evolution are extremely labor-intensive activities, a great deal of productivity leverage can be gained by making software production a more capital-intensive activity. Typically, capital investment per software worker has been little different from the \$2000–\$3000 per person typical of office workers in general. However, a number of organizations such as Xerox, TRW, IBM, and Bell Laboratories have indicated that significantly higher investments (in the \$10,000–\$30,000 per person range) have been more than recaptured in improved software productivity.

Providing software personnel with private offices is another cost-effective measure, leading to productivity gains of roughly 11 percent at IBM-Santa Teresa (see Jones[2]) and 8 percent at TRW (Boehm et al.[3]). Similar results on the payoffs of capital investments in better facilities and support capabilities have been reported in other studies (see "Further Reading").

Management. Poor management can decrease software productivity more rapidly than any other factor. Here are some examples of the major classes of management activities that most frequently contribute to losses in productivity:

• *Poor planning.* An example of poor planning was a project with very vague test plans. When the 20-person test team came on board, they found no test data, test drivers, test facilities, test strategies and procedures, or test readiness standards for the developers' code. As a result, the project incurred a 30 percent overrun in cost and a 40 percent overrun in schedule.

• *Poor skill mix.* Poor skill mix is often a result of the Peter Principle: "In a hierarchy, every employee tends to rise to his level of incompetence." The most common realization of the Peter Principle in software engineering is the practice of "advancing" good programmers by promoting them into management. Sometimes this works well, but overall it produces more mismatches, frustration, and dam-aged careers in software engineering than in other fields. This point has been realized by a number of organizations, which have instituted dual or multiple career paths culminating in "superprogrammer" or "superanalyst" as well as "supermanager."

• *Premature staffing.* An example of premature staffing is the following quotation from a small-project manager: "At an early stage in the design, I was made the project manager and given three trainees to help out on the project. My biggest mistake was to burn up half of my time and the other senior designer's time trying to keep the trainees busy. As a result, we left big holes in the design, which killed us in the end. " A related source of decreased productivity is the attempt to speed up a project by adding more people, in contradiction to Brooks's Law: "Adding more people to a late software project will make it later."

• *Premature coding.* An example of premature coding is the WISCA syndrome, where WISCA stands for "Why isn't Sam coding anything?" A counterpart is the statement, "We'd better hurry up and start coding, because we're going to have a lot of debugging to do." The most important management property of an efficient multiperson software development is the achievement of a set of thorough, validated, and stable module interface specifications, which allow the developers to operate in parallel without being swamped by interpersonal communications overhead. As early as 1961, software managers were realizing that "every sheet of accurate inter-face definition is, quite literally, worth its weight in gold."

• *Poor reward structure.* An example of poor reward structure is the organization that gives its top performers 6 percent raises and its mediocre performers 5 percent raises. Eventually, the good people get frustrated and leave. A great deal can be done by creative application of other rewards, such as special bonuses, grade-level promotions, travel and special courses, and recognition programs for top performers.

Making steps more efficient. The value chain in Figure 4 provides a basic set of insights on the relative productivity leverage involved in eliminating or improving the efficiency of the various steps in the software process. For example, since the process of performing code and unit test functions consumes only eight percent of the software life-cycle dollar, the productivity impact of tools to eliminate code and unit test or to make it more efficient will not exceed eight percent (unless the tools also eliminate other classes of effort, such as rework in later phases).

The primary leverage factor in making the existing software process steps more efficient is the use of software tools to automate the current repetitive and labor-intensive portions of each step.

Experience to date suggests that software tools are much more effective if they are part of an integrated project support environment (IPSE). The primary features that distinguish an IPSE from an ad hoc collection of tools are

- a set of common assumptions about the software process model being supported by the tools (or. more strongly. a particular software development method being sup-ported by the tools);
- an integrated project master database or persistent object base serving as a unified repository of the technical and management entities created during the software process. along with their various versions, attributes, and relationships;
- support of the entire range of users and activities involved in the software project. not just of programmers developing code;
- a unified user interface providing easy and natural ways for various classes of project personnel (expert programmers. novice librarians.

secretaries. managers. planning and control personnel. etc.) to draw on the tools in the IPSE;

- a critical-mass ensemble of tools. covering significant portions of software project activities;
- a computer communication architecture facilitating user access to data and resources in the IPSE.

Eliminating Step. A good many automated aids go beyond simply making steps more efficient, to the point of fully eliminating previous manual steps. If we compare software development today with its counterpart in the 1950’s, we see that assemblers and compilers are excellent examples of ways to vastly improve productivity by eliminating steps. More recent examples are process construction systems, software standards checkers and other quality assurance functions, and requirements and design consistency checkers.

More ambitious efforts to eliminate steps involve the automation of the entire programming process by providing capabilities that operate directly on a set of software specifications to automatically generate computer programs. There are two major branches to this approach: domain-specific and domain-independent automatic programming.

The domain-specific approach gains advantages by capitalizing on domain knowledge in transforming specifications into programs and in constraining the universe of programming discourse to a relatively smaller domain. In the limit, one reaches the boundary with fourth-generation languages such as Visicalc, which are excellent automatic programming systems within a very narrow domain and relatively ineffective outside that domain.

The domain-independent approach offers a much broader payoff in the long run but has more difficulty in achieving efficient implementations of larger-scale programs.

Eliminating Rework. The strongest opportunity identified by the value chain analysis in Figure 4 is the 30 percent productivity leverage available through eliminating rework. Actually, the leverage factor is probably more like 50 percent over the life cycle. since most of the sources of rework savings (e.g., modern programming practices and rapid prototyping) will reduce the incidence of current postdevelopment software modifications (e.g., to fix residual errors or to finally get the requirements right) as well as making the modifications more efficient.

The major rework opportunity areas identified in the opportunity tree in Figure 5 are front-end aids; knowledge-based software assistants; information hiding and modern programming practices, incremental development, improved process models, and rapid prototyping. (In addition, reusing components can significantly reduce rework.)

Front-End Aids. Software computer-aided design and requirements analysis tools can eliminate a great deal of rework through better visualization of software requirements and design specification, more formal and unambiguous specifications, automated consistency and completeness checking, and automated traceability of requirements to design. Probably the most extensive of these systems is the Distributed Computing Design System, which includes a system specification language, a software requirements specification language, a distributed-system design language, and a module description language. A number of commercial front-end aids are also available such as ISDOS?PSI-PSA, SADT, CASE, Excelerator, IDE, Cadre, and Ada Graph. Some complementary front-end aids include rapid simulation aids such as RSA and executable specification aids such as Paisley.

Knowledge-Based Software Assistants. In many application areas, the artificial intelligence community is finding that total automation of knowledge-intensive functions falls in the "currently too hard" category but that combinations of conventional and AI techniques may he used to provide useful automated assistance to human experts in performing complex tasks. This is the primary motivation for the knowledge-based software assistant (KBSA) concept, as described by Green et al.[6]

The primary benefit of a KBSA will he the elimination of much of the rework currently experienced on software projects due to the belated appreciation that a previous programming or project decision was inappropriate. A number of prototype KBSAs are currently under development in such areas as acquisition management, configuration management, problem report tracking, algorithm selection, data structuring, choice of reusable components, and project planning and control.

Information hiding and other modern programming practices. In general, modern programming practices (MPPs) such as early verification and validation, modular design, top-down development, structured programming, walk-throughs or inspections, and software quality standards achieve their productivity leverage through avoidance of rework. As indicated in Figure 3, MPPs provide a productivity range of 1.51 during development and up to 1.92 for the life cycle of a large software product.

A particularly powerful technique for eliminating rework is the information-hiding approach developed by Parnas and applied in the US Navy A-7 project (Parnas, Clements, and Weiss).[7] This approach minimizes rework by hiding implementation decisions within modules, thus minimizing the ripple effects usually encountered when software implementation decisions need to be changed. The information-hiding approach can be particularly effective in eliminating rework during software evolution, by identifying the portions of the software most likely to undergo change (characteristics of workstations, input data formats, etc.) and hiding these sources of evolutionary change within modules.

As an example, the current requirements may specify that a particular user workstation or terminal is to be used. By also identifying in the requirements the terminal characteristics most likely to change (line width, character set, access protocols, etc.), the designers can hide these details of the terminal inside a terminal-handler module, thus isolating the remainder of the software from the usual ripple effects accompanying a change in the terminal characteristics.

This approach revolutionizes the concept of a requirements specification. Rather than being just a snapshot of a system's software requirements at a single point in time, the requirements specification must also identify the most likely requirements evolution paths the system will experience. This also means that a design validation activity should address not just traceability to the current requirements snapshot but also how well the design accommodates the expected directions of change.

Modern programming practices and Ada. A major initiative to embody modern programming practices and information hiding concepts into standard programming practice has been the U.S. Department of Defense's development of the programming language Ada. Ada has constructs such as packages that support modularity, information hiding, and reuse; strong typing that avoids rework due to common programming errors; structured programming constructs; and a number of other advanced features addressing such issues as concurrency, exception handling, and generic programs. Getting all of these features to work together has strained the state of the art in compiler development, but currently a number of effective Ada compilers are available.

Assessing the likely productivity impact of Ada is difficult because the Ada concept is also intended to include an overall programming and project environment and because most Ada capabilities are not yet fully mature. However, several studies have estimated the comparative life-cycle productivity of an Ada project and a conventional HOL project as a function of time, using as parameters the cost driven variables and productivity ranges of models such as COCOMO. The typical result of these studies has indicated an initial added cost of 12–30 percent for initial uses of ada, a breakeven point in the 1988–1989 time frame, and a long-range savings of 40–50 percent for a fully mature Ada support environment and development staff.

Improved process models. The leading current model of the software process, the waterfall model,[1] tends to focus a software project toward the production of a series of documents (system specification, software requirements specification, top-level design document, and detailed design document). When used in concert with thorough front-end validation activities, the waterfall approach is very effective in reducing rework. But, frequently, the document-driven interpretation of the waterfall model pushes a project toward more rapid production of documents rather than toward thinking through critical issues. For example, a recently proposed government software progress reporting scheme focuses on the number of unresolved elements in the software requirements and design specifications. If a project manager wants to show rapid progress, he is actually tempted to work on resolving the easy elements rather than the hard ones on to complete the document quickly by putting in arbitrary rather than well-reasoned specifications.

An important point in this regard is that rework instances tend to follow a Pareto distribution: 80 percent of the rework costs typically result from 20 percent of the problems. Figure 6 shows some typical distributions from recent TRW software projects; similar trends have been indicated in other studies. The major implication of this distribution is that software verification and validation activities should focus on identifying and eliminating the specific high-risk problems to be encountered by a software project, rather than spreading the early problem elimination effort uniformly across trivial and severe problems. Even more strongly, this implies that a risk-driven approach to the software life cycle such as the spiral model (see Boehm[8]) is preferable to a more document-driven model such as the traditional waterfall model. The spiral model organizes the software development process into a sequence of increasingly detailed definition cycles. The amount of emphasis in each cycle on documentation, simulation, prototyping, or other definition activities is determined by the relative risk of not resolving key definition issues. Thus, the spinal model focuses effort on identification and early resolution on the 20 percent of the problems that will otherwise account for 80 percent of the rework costs.

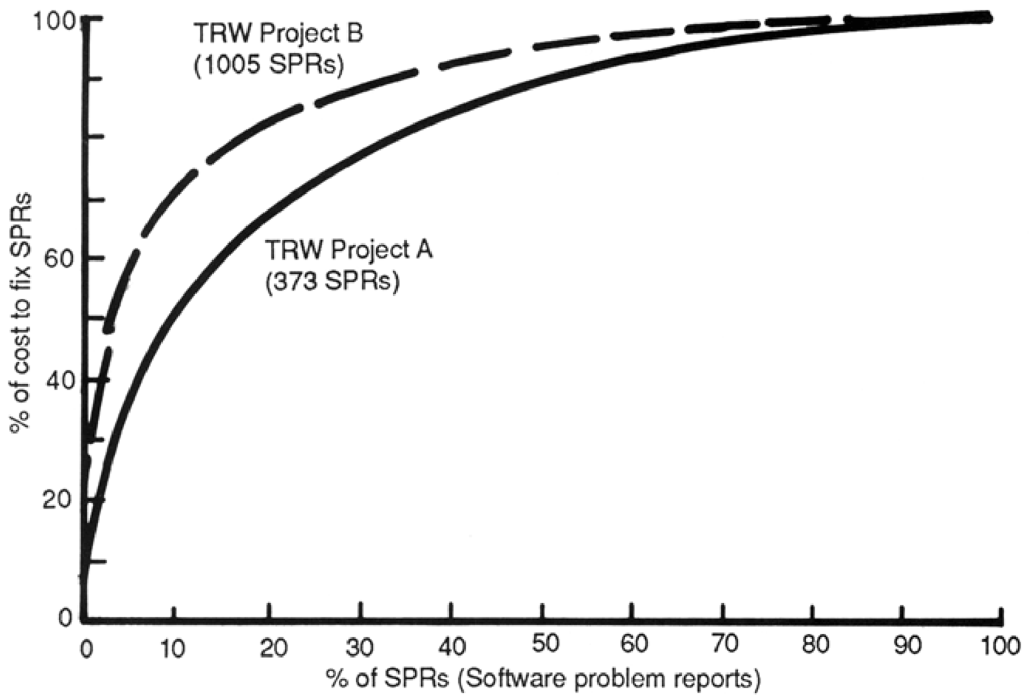


Figure 6. Rework costs are concentrated in a few high-risk items.

Rapid Prototyping. One of the major sources of rework found in the data represented by Figure 4 were portions of a software specification based on poorly understood mission on user interface requirements. A primary example is the user who says, “I can’t tell you exactly what I want, but I’ll know it when I see it.” A number of rapid prototyping aids have become available to improve this situation. A good many are based on the interpretive-execution capabilities of advanced artificial intelligence environments such as Interlisp. Others are based on two-phase interactive-graphics composition and execution capabilities using conventional HOLs. Other rapid prototyping systems provide risk reduction capabilities for rapid assessment of real-time performance issues on distributed data processing issues.

Building Simpler Products. As indicated in Figure 3, the largest productivity range available to the software develop-en comes from the number of instructions one chooses to develop. There are two primary options here: one is building simpler products; the other is reusing software components.

Besides their contribution to eliminating rework, the last two approaches involving rapid prototyping and improved software process models can also be very effective in improving bottom-line productivity by building simpler products to eliminate software gold plating: extra software that not only consumes extra effort but also reduces the conceptual integrity of the product.

For example, a recent seven-project experiment comparing a specification-oriented approach and a prototyping-oriented approach to the development of small-user-intensive application software products (see Boehm, Gray, and Seewaldt[9]) is illustrated in Figure 7. The experiment found primarily that

- on the average ($\overline{\mathrm{P}}$ vs $\overline{\mathrm{S}}$ in Figure 7), both approaches resulted in roughly equivalent productivity in delivered source instructions pen man-hour (DSI/MH);
- the prototyping projects developed products with roughly equivalent performance but requiring roughly 40 percent fewer DSI and 40 percent fewer man-hours;
- the specifying projects had less difficulty in debugging and integration due to their development of good interface specifications.

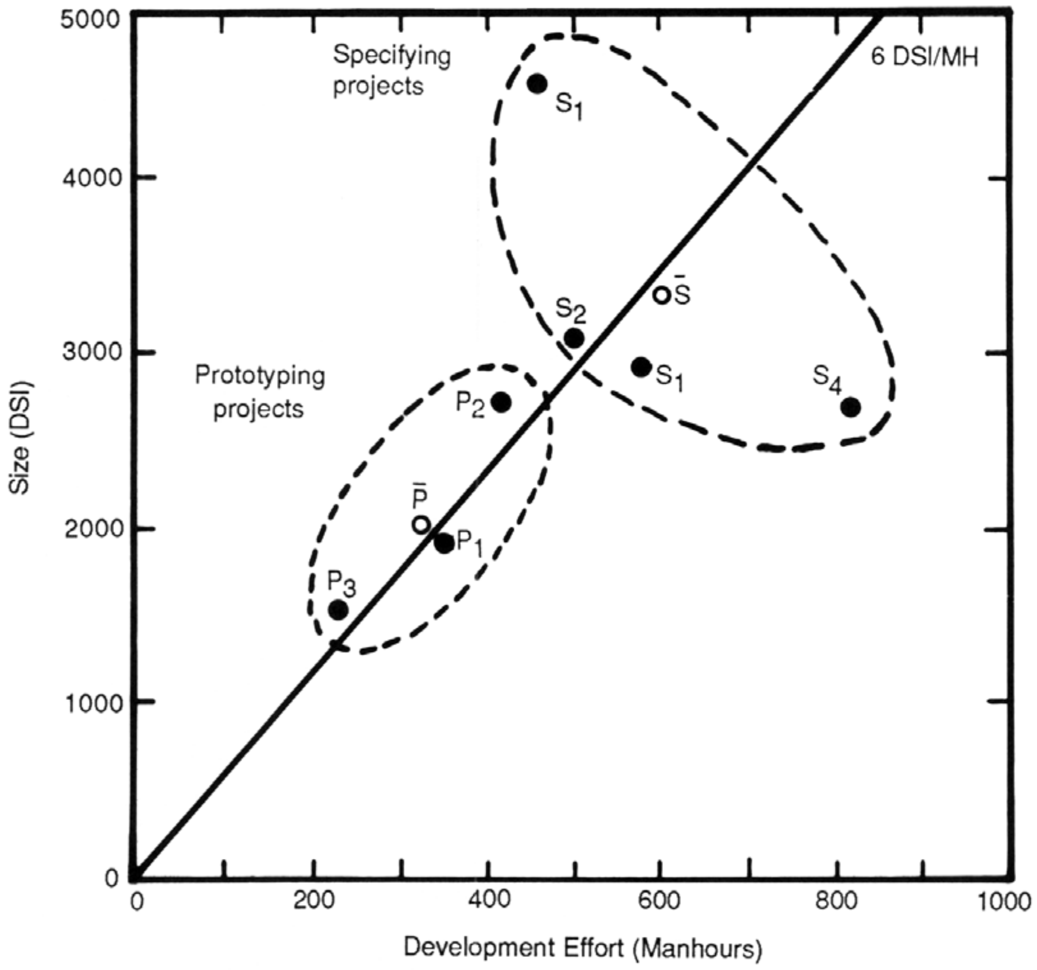


Figure 7. Prototyping versus specifying experiment: size and effort comparisons.

The final point indicates that prototypes are not a panacea for all problems and that specifications are still very important. However, one of the telling insights in this experiment was the comment of one of the participants using the specification approach: "Words are cheap." During the specification phase, it is all too easy to add gold-plating functions to the product specification, without a good understanding of their effect on the product's conceptual integrity or the project's required effort. As Heckel[10] writes:

> Most programmers … defend their use of a software feature by saying, "You don't have to use it if you don't want to, so what harm can it do?" It can do a great deal of harm. The user might spend time trying to understand the feature, only to decide it isn't needed, or he may accidentally use the feature and not know what has happened or how to get out of the mistake. If a feature is inconsistent with the rest of the user

> interface, the user might draw false conclusions about the other commands. The feature must be documented, which makes the user's manual thicker. The cumulative effect of such features is to overwhelm the user and obscure communication with your program…

A further discussion of typical sources of software gold plating and an approach for evaluating potential gold-plating features is provided by Boehm[1], Chapter 11.

Some of the newer software process models stimulate the development of simpler products. One of the difficulties of the traditional waterfall model is that its specification-driven approach can frequently lead one along the "Words are cheap" road toward gold-plated products, as discussed above. A frequent experience in the specification of large systems is that users with little feel for a computer system will overspecify on functionality and performance just to make sure the system will include what they need.

The evolutionary development model (see McCracken and Jackson[11]) emphasizes the use of prototyping capabilities to converge on the necessary or high-leverage software product features essential to the user's mission. The related transformational model (see Balzer, Cheatham, and Green[12]) shortcuts the problem by providing (where available) a direct transformation from specification to executing code, thus supporting both a specification-based and an evolutionary-development approach. The spiral model discussed above addresses the gold-plating problem by focusing on a continuing determination of users' mission objectives and a continuing cost-benefit analysis of candidate software product features in terms of their contribution to mission objectives.

Reusing components. Another key to improving productivity by writing less code is the reuse of existing software components. The simplest approach in this direction involves the development and use of libraries of software components.* A great deal of progress has been made in this direction, particularly in such areas as mathematical and statistical routines and operating-system-related utilities. Further progress is possible via similar capabilities in user application areas. For example, Raytheon's library system of reusable business application components has achieved typical figures of 60 percent reusable code for new applications, and typical cost savings of 10 percent in the design phase, 50 percent in the code and test phase, and 60 percent in the maintenance phase. Toshiba's system of reusable components for industrial process control has resulted in typical productivity rates of over 2000 source instructions pen man-month for high-quality industrial software products.

* There is a good deal of productivity leverage in reusing software specifications, designs, and plans, as well as code.

At this level of sophistication, such systems would better be called application genera-tons, rather than component libraries, because they have addressed several system-oriented component compatibility issues such as component interface conventions, data structuring, and program control and error handling conventions. Similar characteristics have made Unix a strong foundation for developing application generators.

One can proceed even further in this direction to create a very high level language or fourth-generation language (4GL) by adding a language for specifying desired applications and a set of capabilities for interpreting user specifications, configuring the appropriate set of components, and executing the resulting program. Currently, the most fertile areas for 4GLs are spreadsheet calculators (Visicalc, Multiplan, I -2-3, etc.) and small-business systems typically featuring a DBMS, report generator, database query language, and graphics package (Nomad, Ramis, Focus, ADF, DBase II, etc.).

Some 4GL advocates promise factors of 10 to 100 improvement in productivity from the use of4GLs. Are such factors achievable?

The best experimental evidence on the productivity leverage of 4GLs is provided by a six-project experiment comparing the use of a third-generation programming language (Cobol) and a fourth-generation language (Focus) on a mix of small business-application projects involving both experts and beginners developing both simple and complex applications (see Harel and McLean[13]). Its primary findings, illustrated in Figure 8, are summarized as follows:

- On an overall average (the $\overline{C}$ and the $\overline{F}$ in Figure 8), the fourth-generation approach produced equivalent products to the third-generation approach, with about 60 percent fewer DSI and 60 percent fewer man-hours (again with roughly equivalent productivity in DSI/MH).
- From project to project, there was a significant variation in the ratio of third-generation to fourth-generation DSI (0.9:1 to 27:1), man-hours (1.5:1 to 8:1), and DSI/MH (0.5:1 to 5:1).

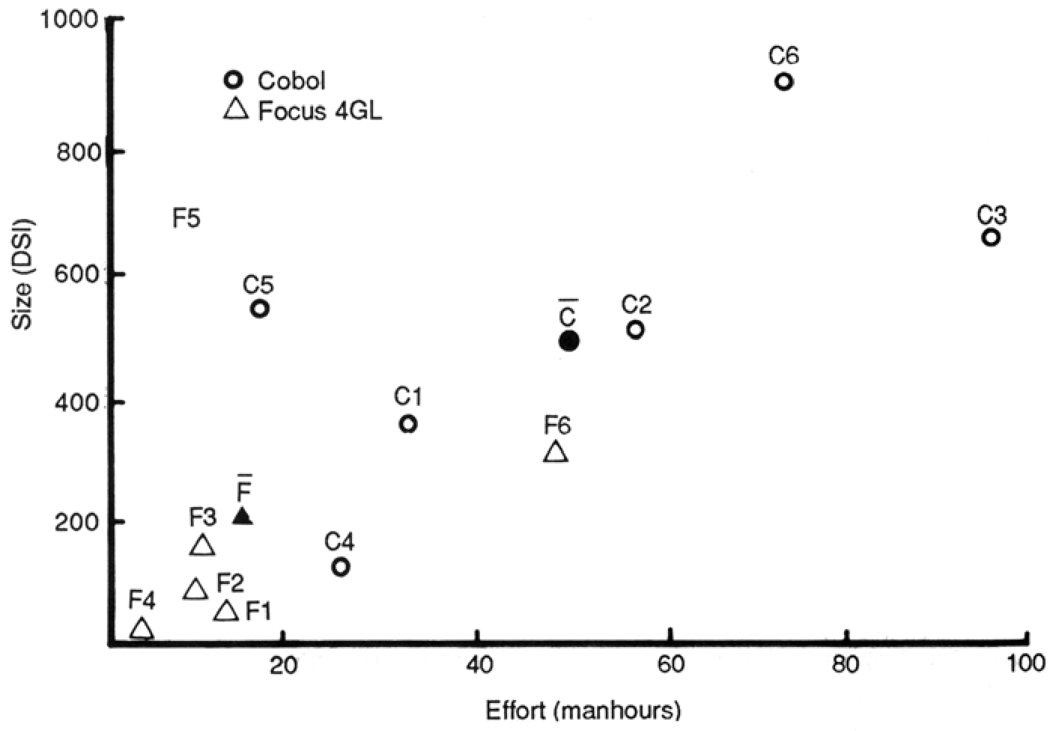


Figure 8. Fourth-generation-language experiment: size and effort comparisons.

Although the average Cobol effort was 2.5 times higher than the average Focus effort for the same application, the effect is fan from uniform across a spectrum of applications. Thus, it is difficult to predict the 4GL productivity gain for any particular application.

Guimaraes[14] provides further evidence from a survey of 43 organizations that 4GLs reduce personnel costs, reduce user frustration, and more quickly satisfy user information needs within their domain of applicability. On the other hand, the survey found 4GLs extremely inefficient of computer resources and difficult to interface with conventional applications programs. Some major disasters have occurred in attempting to apply purely 4GL solutions to large, high-performance applications such as the New Jersey motor vehicle registration system[15].

Overall, though, 4GLs offer an extremely attractive option for significantly improving software productivity, and attempts are underway to create 4GL capabilities for other application areas. Short of a 4GL capability, the other more limited approaches to reusability such as component libraries and application generators can generate near-term cost savings and serve as a foundation for more ambitious 4GL capabilities in the long run.

Software productivity trends

It is difficult to summarize such a welter of issues as those involved in improving software productivity. Figure 9 provides at least a partial summary of some of the key lever-age areas. It shows our typical overall progress in improving software productivity between the 1960–65 era and the 1980–85 era, in terms of equivalent machine instructions per man-month, and it projects our potential progress by the 1995–2000 era.

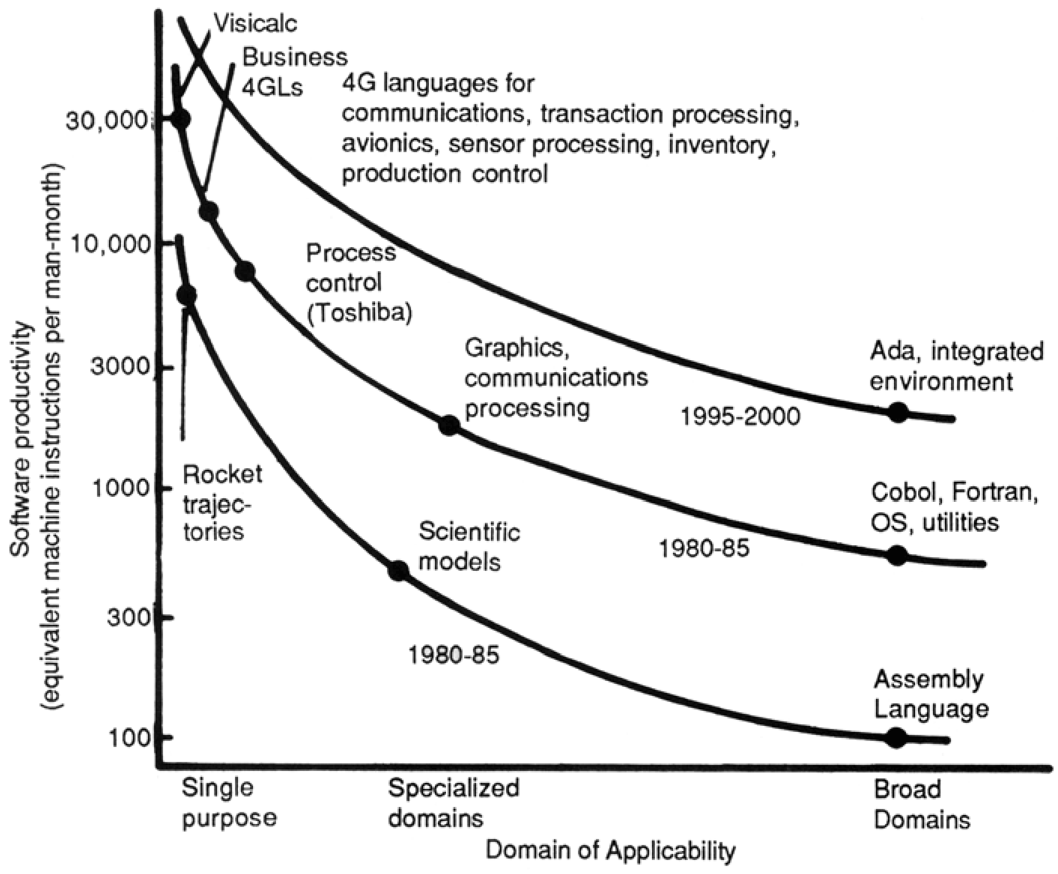


Figure 9. Software technology and productivity trends.

The horizontal dimension in Figure 9 is a qualitative scale indicating the breadth of the domain of applicability of a given software productivity capability. It reflects the fact that our most impressive software productivity achievements to date have been made by exploiting our knowledge of particular application domains.

Thus, for example, even in the early 1960's, when most large, general-purpose systems were being developed in assembly language at a typical rate of

100 delivered machine instructions per man-month (DMI/MM), there were application generators providing productivity rates of 3000 DMI/MM and higher. Several examples were available in the area of rocket trajectory computation, where systems such as Rocket (see Boehm[16]) provided a library of reusable components (for aerodynamics, propulsion, guidance and control, earth models, etc.) and specialized extensions of FORTRAN for users to specify how to link components together to simulate their desired multistage rocket vehicle and flight program.

By the early 1980's, we had progressed in the power and range of domain-specific application generators, so that even higher productivity figures were being achieved in such a variety of domains as spreadsheet calculations (30,000 DMI/MM and up), industrial process control, and business fourth-generation languages. At the same time, we had extended our general-purpose capabilities from assembly language and primitive batch operating systems to HOLs with collections of tools providing on the order of 600 DMI/MM for large, broad-domain applications. According to Brooks[17], those capabilities address the elimination of the "accidental" difficulties in developing software. The domain-specific capabilities shown on the left side of Figure 9 are aimed at reducing the "essential" portion of software acquisition costs.

Thus, for the 1995–2000 time frame, we can see that two major classes of opportunities for improving software productivity exist: providing better support systems for broad-domain applications, involving fully integrated methods, environments, and modern programming languages such as Ada; and extending the number and size of the domains for which we can use domain-specific fourth-generation languages and application genera-tons. Examples of promising future application domains include communications processing, transaction processing, sensor data processing, broaden process control areas such as avionics and job shop production control, and broader DBMS-oriented areas such as inventory control and production management.

We have seen that the magnitude and continuing growth of software costs create a strong need to improve software productivity. This implies a need to carefully define software productivity, and since our current productivity metrics are not fully satisfactory, to work on better ones. It also implies a need to develop capabilities that improve not only software productivity but also software quality.

The analyses of software productivity ranges and the software value chain led to the definition of a software productivity opportunity tree that identifies the major opportunity areas for improving productivity:

- Getting the best from people via better management, staffing, incentives, and work environments.

- Developing and using integrated project-support environments, which automate portions of the development and evolution process and make them more efficient.
- Eliminating rework via better front-end aids, risk management, prototyping, mere-mental development, and modern programming practices, particularly information hiding.
- Writing less code by reusing software components, developing and using application generators and fourth-generation languages, and avoiding software gold plating.

As a final conclusion, one point deserves particular emphasis. In pursuing improvements in software productivity, we need to be careful not to confuse means with ends. Improved software productivity is not an end in itself; it is a means of helping people better expand their capabilities to deal with information and to make decisions. Often, helping people to do this will involve us in activities (for example, spending two weeks helping someone find an effective nonsoftware solution to a problem) that do not add points to our software productivity scoreboard. At such times, we need to recall that the software productivity scoreboard is just one of the many ways we have to gauge our progress to-ward better use of computers to serve people.

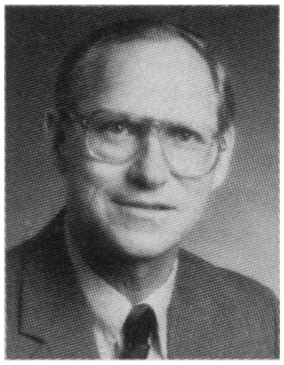
Acknowledgments

I would like to thank Frank Belz, Phil Papaccio, Lol Penedo, and George Spadaro of TRW, and Bill Riddle of the Software Productivity Consortium, for substantial assistance in refining earlier versions of this article, and Bruce Shriven and the *Computer* reviewers for a particularly thorough and helpful job.

References

1. B.W. Boehm, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, N.J., 1981.
2. T.C. Jones, *Programming Productivity,* McGraw-Hill, New York, 1986.
3. B.W. Boehm et al., “A Software Development Environment for Improving Productivity,” *Computer*, Vol. 17, No. 6, June 1984, 30–44.
4. M.E. Porter, *Competitive Advantage*, Free Press, New York, 1985.
5. C.E. Walston and C.P. Felix, “A Method of Programming Measurement and Estimation,” *IBM Systems J.*, Vol. 16, No. 1, 1977, pp. 54–73.
6. C.C. Green et al., “Report on a Knowledge-Based Software Assistant,” USAF/RADC Report RADC-TR-l95, Aug. 1983.

7. D.L. Parnas, P.C. Clements, and D.M. Weiss, “The Modular Structure of Complex Systems,” *IEEE Trans. Software Engineering*, Mar. 1985, pp. 259–266.
8. B.W. Boehm, “A Spiral Model of Software Development and Enhancement,” *Proc. IEEE Second Software Process Workshop, ACM Software Engineering Notes,* Aug. 1986.
9. B.W. Boehm, T.E. Gray, and T. Seewaldt, “Prototyping vs. Specifying: A Multi-Project Experiment,” *IEEE Trans. Software Engineering*, May 1984, pp. 133–145.
10. P. Heckel, *The Elements of Friendly Software Design*, Warner Books, 1984.
11. D.D. McCracken and M.A. Jackson, “Life Cycle Concept Considered Harmful,” *Software Engineering Notes*, *ACM*, Apr. 1982, pp. 29–32.
12. R. Balzer, T.E. Cheatham, and C. Green, “Software Technology in the 1990’s: Using the New Paradigm,” *Computer*, Vol. 16, No. 11, Nov. 1983, pp. 39–45.
13. E. Harel and E.R. McLean, “The Effects of Using a Nonprocedural Language on Programmer Productivity,” UCLA Graduate School of Management, Information Systems Working Paper #3-83, Nov. 1982.
14. T. Guimaraes, “A Study of Application Program Development Techniques,” *Comm. ACM*, May 1985, pp. 494–499.
15. C. Babcock, “New Jersey Motorists in Software Jam,” *Computerworld,* Sept. 30, 1985, pp. 1, 6.
16. B.W. Boehm, *ROCKET: Rand*’s *Omnibus Calculator of the Kinematics of Earth Trajectories*, Prentice-Hall, Englewood Cliffs, N.J., 1964.
17. F.P. Brooks, “No Silver Bullet—Essence and Accidents of Software Engineering,” *Proc. IFIP Congress 1986*, North Holland, 1986, pp. 1069–1076. Also in *Computer*, April 1987, pp. 10–19.



Barry W. Boehm is the chief scientist in the Office of Technology at the TRW Defense Systems Group. He is responsible for the group's Ada Office, Technology Education Program, and Quantum Leap Software Development

Program. Since joining TRW in 1973, Boehm has held several senior engineering and management positions. He is also an adjunct professor at UCLA and a member of the Computer Society's Board of Governors. Boehm received his BA in mathematics from Harvard in 1957 and his MA and PhD from UCLA in 1961 and 1964.

Further reading

Each of the topics in this article involves a fascinating complex of ideas and capabilities that could only be summarized. Listed below are additional articles and books that amplify some of its specific points or cover one of its main topics in more detail.

Software cost drivers and trends

Boehm, B.W., “Understanding and Controlling Software Costs,” *Proc. IFIP Congress 1986*, North Holland, 1986, pp. 703–714. An updated version to appear in *IEEE Trans. Software Engineering*, 1987.

Boehm, B.W., *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, N.J., 1981.

Brooks, F.P., No Silver Bullet—Essence and Accidents of Software Engineering, in *Proceedings ol IFIP Congress /986,* North Holland, 1986, pp. 1069–1076. Also in *Computer*, Apr. 1987, pp. 10–19.

DeMarco, T., *Controlling Software Project,* Yourdon, New York, 1982.

Jones, T.C., “Demographic and Technical Trends in the Computing Industry,” Software Productivity Research, Inc., July 1983.

Jones, T.C., *Programming Productivity,* McGraw-Hill, New York, I 986.

Walston, C.E., and C.P. Felix, “A Method of Programming Measurement and Estimation,” *IBM Systems J.,* Vol. 16, No. 1, 1977, pp. 54–73.

Wegner, P., “Capital-Intensive Software Technology,” *IEEE Software*, Vol. 1, No. 3, 1984, pp. 7–45.

Software Productivity and Quality Metrics

Basili, V., *Tutorial on Models and Metrics for So/tware Management and Engineering,*. Computer Society Press, Los Alamitos, Calif., 1980.

Boehm, B.W., *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, N.J., 1981.

Boehm, B.W., et al., *Characteristics* of *Software Quality,* North Holland, 1978.

Bowen, T.P., G.B. Wigle, and J.T. Tsai, "Specification of Software Quality Attributes," RADC-TR85-37 (3 vol.), Feb. 1985.

Conte, S.D., H. Dunsmore. and V. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, 1986.

DeMareo, T., *Controlling Software Projects,* Yourdon, New York, 1982.

Frewin, E., et al., "Quality Measurement and Modeling—State of the Art Report," ESPRIT Report REQUEST/STC-gdf/001/51/QL-RP/00.7, July 1985.

Gilb, T., *Principles of Software Engineering Management*, Addison-Wesley. 1987.

Grady. R., and D. Caswell, *Software Metrics: Establishing a Company-Wide Program,* Prentice-Hall, 1987.

"GUIDE Survey of New Programming Technologies," *GUIDE 1979 Proceedings* Guide. Inc.. 1979, pp. 306-308.

Halstead, M.H., *Elements of Software Science,* Elsevier, New York, 1977.

Jones, T.C., *Programming Productivity,* McGraw-Hill, New York, 1986.

McCabe, T.C. "A Complexity Measure," *IEEE Trans. Software Engineering,* Dec. 1976, pp. 308–320.

Thadhani, A.J., "Factors Affecting Programmer Productivity During Application Development," *IBM Systems J.,* Vol. 23, Nov. 1984, pp. 19–35.

Weinberg. G.M., and F.L. Schulman, "Goals and Performance in Computer Programming," *Human factors,* Vol. 16, No. 1, 1974, pp. 70–77.

Getting the best from people

Brooks. F.P., *The Mythical Man-Month,* Addison-Wesley, Reading, Mass., 1975.

Couger, J.D.. and R.A. Zasvacki. *Motivating and Managing Computer Personnel,* Wiley. New York, 1980.

Curtis, B., *Human Factory in Software Development,* IEEE Cat. No. FHO I 85–9. 1981.

DeMarco, T., *Controlling Software Projects,* Yourdon, New York, 1982.

DeMarco, T., and T. Lister, "Programmer Performance and the Effects of the Workplace," *Proc. Eighth Int'l Conf. Software Engineering* Aug. 1985, pp. 268–272.

Hosier, W.A., "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," *IRE Trans. Engineering Management*, June 1961, pp. 99–115.

Manley, J.H., "Software Engineering Provisioning Process," *Proc. Eighth Int'l Conf. on Software Engineering*, Aug. 1985, pp. 273–284.

Metzger, P.J.. *Managing a Programming Project*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ., 1981.

Weinberg, G.M., *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.

Weinberg, G.M., and E.I. Schulman. "Goals and Performance in Computer Programming," *Human Factory,* Vol. 16, No. 1, 1974, pp. 70–77.

Yourdon, E., *Managing the Structured Techniques,* 3rd ed., Yourdon Press, 1987.

Making steps more efficient

Alford, M.W., "SREM at the Age of Eight: The Distributed Computing Design System," *Computer*, Vol. 18, No. 4, Apr. 1985.

Boehm, B.W., et al., "A Software Development Environment for Improving Productivity," *Computer,* Vol. 17. No. 6, June 1984, pp 30–44.

Green, C.C., et al., "Report on a Knowledge-based Software Assistant," USAF/RADC Report RADC-TR-l95, Aug. 1983.

Henderson, P., ed., *Proc. Second Conf. Practical Software Development Environments, ACM SIGPLAN Notices,* Dec. 1986.

Houghton, R. C., "Software Development Tools," National Bureau of Standards, NBS Special Report 500-88, 1982.

Sommerville, I., ed., *Software Engineering Environments,* Peter Peregrinus, Ltd., 1987.

Wegner, P., "Capital-Intensive Software Technology," *IEEE Software*, Vol. 1, No. 3, July 1984, pp. 7–45.

Eliminating steps

Barstow, D.R., "Artificial Intelligence and Software Engineering," *Proc. ICSE 9,* IEEE, March 1987.

"Special Issue on Automatic Programming," *IEEE Trans. Software Engineering,* Nov. 1985.

Eliminating rework

Alford, M.W., "SREM at the Age of Eight: The Distributed Computing Design System," *Computer*, Vol. 18, No. 4, Apr. 1985.

Boar, B.H., *Application Prototyping,* John Wiley, New York, 1984.

Boehm, B.W., "A Spiral Model of Software Development and Enhancement," *Proc. IEEE Second Software Process Workshop, ACM Software Engineering Notes,* Aug. 1986.

Boehm, B.W., "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software,* Vol. 1, No. 1, Jan. 1984, pp. 75–88.

Booch, G., *Software Engineering with Ada*, Benjamin/Cummings, 1983.

Mills, H.D., "Structured Programming: Retrospect and Prospect," *IEEE Software*, Nov. 1986, pp. 58–66.

Parnas, D.L., "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Software Engineering*, Mar. 1979, pp. 128–137.

Swinson, G.E., "Workstation-Based Rapid Simulation Aids for Distributed Processing Networks," *Proc. IEEE Simulation Conference*, 1984.

Yourdon, E., *Managing the Structured Techniques,* 3rd ed., Yourdon Press, 1987.

Zave, P., “The Operational Versus the Conventional Approach to Software Development,” *Comm. ACM,* 104–118, Feb. 1984.

Zelkowitz, M., and S. Squires, eds., *Proc. ACM Rapid Prototyping Svmp.*, ACM, Oct. 1982.

Building simpler products

Agresti, W., *New Paradigms, for Software Development,* IEEE Cat. No. EH 0245-1, 1986.

Dowson, M., and J.C. Wileden, eds., *Proc. Second Software Process Workshop, ACM Software Engineering Notes,* Aug. 1986.

Heckel, P., *The Elements of Friendly Software Design,* Warner Books, 1984.

Lehman, M.M., V. Stenning, and C. Potts, eds., *Proc. Software Process Workshop,* IEEE, Feb. 1984.

McCracken, D.D., and M.A. Jackson, “Life Cycle Concept Considered Harmful,” *Software Engineering Notes, ACM,* Apr. 1982, pp. 29–32.

Reusing components

Boehm, B.W., *Software Engineering Economics* (Chapter 33), Prentice-Hall, Englewood Cliffs, NJ, 1981.

Jones, T.C., *Programming Productivity,* McGraw-Hill, New York, 1986.

Freeman, P., *Software Reusability,* IEEE Cat. No. EH0256-8, 1987.

Horowitz, E., A. Kemper, and B. Narasimhan, “A Survey of Application Generators,” *IEEE Software*, Jan. 1985, pp. 40–54.

Lanergan, R.G., and C.A. Grasso, “Software Engineering with Reusable Design and Code,” *IEEE Trans. Software Engineering*, Sept. 1984, pp. 498–501.

Matsumoto, Y., “Management of Industrial Software Production,” *Computer*, Vol. 17, No. 2, 1984, pp. 59–70.

“Special Issue on Reusability in Programming,” *IEEE Trans. Software Engineering*, Sept. 1984.