SOFTWARE QUALITY

Concepts and Plans

SOFTWARE QUALITY

Concepts and Plans

Robert H. Dunn
Systems for Quality Software
Management Consultants

PRENTICE HALL
Englewood Cliffs, N.J. 07632

Preface

Supposedly, Haydn got the idea for the opening theme of one of his quartets from the singing of a lovesick lark. We have all heard that a falling apple inspired Newton's work on gravitation. As viewed by either producers or consumers, a few software successes have happened without anyone giving much thought to quality. All of which demonstrate that serendipity does work every now and again.

This book is written for those who cannot depend on accident, but need a deliberate path to software quality. To this end, the book deals with planning the measures one can take in the interests of software quality. Part V, in fact, is given to formal software quality plans, Parts I through IV deal with basic concepts that should lie beneath quality plans, not least of which is the business of defining what we mean by software quality. (Section 1.5 of the first chapter amplifies this skimpy outline.)

Let us identify the people who demand constructive approaches to software. Certainly, software managers, but also programmers who aspire to management. Another group of people consists of the ever-increasing number of systems engineers and managers attacking—or attacked by—software-intensive projects. An obvious audience for the book is made up of quality control (or quality assurance) managers involved with software-intensive systems or concerned about the quality of the software their companies depend on. We need to include one more category, the one that lies at the intersection of quality assurance and software engineering: software quality engineers.

These are the people the book addresses. (Well, also, my mother-in-law, because she loyally reads everything I write.) Despite the diversity of the intended audience,

 I have tried to accommodate all its parts. The one prerequisite is some exposure to

development or maintenance projects: Terms such as "defect" and "manageable" first appear pages or chapters before their meaning is clarified within the context of software quality.

Admittedly, readers who are not software professionals may find some of the going difficult, especially in Chapter 4, which discusses various software technology issues with respect to the influence of technology on quality or project control. Still, such readers need not give up: The summaries at the end of each chapter provide enough background to permit the reader to skip ahead to more familiar ground.

I should like to thank John Musa, Amrit Goel, and the anonymous reviewers rounded up by Prentice Hall, from whom I received a great many useful suggestions. I should also like to thank my perennial first reader, Steve Dunn, for his usual meticulous criticism of the manuscript.

ROBERT H. DUNN
Buck Hill Road
Easton, Connecticut

Contents

CHAPTER 1

The View from Above

Gather a group of computer scientists or software engineers and ask them to discuss any issues that they care to. Start a stopwatch. In the interval between 7 minutes 30 seconds and 9 minutes 15 seconds (my contribution to the science of software metrics)—after such topics as professional sports, new automobile models, and expectations of interest rate movements have been disposed of—the subject will invariably turn to programmer productivity and software quality.

Gather a group from the ranks of senior management concerned with the development or maintenance of software, and it will take considerably less time to get around to the issues of productivity and quality. They who pay the bills, have to confront disgruntled customers or boards of directors, or have to assume the ultimate risks, tend to get to the bottom line with little delay.

Their risk can be considerable. My last six months as a salaried employee were spent on a project that burned corporate funds at the rate of about $100 million a year in an attempt to adapt a software-intensive line of telecommunications equipment to new markets. The problems faced by the several hundred engineers and programmers were diverse, but nearly all were related to the software of the system. I was only one of many who were convinced that the project was in such disarray that there was no practical way of completing it without spending at least another $250 million, and then only by scrapping much of the work accomplished and starting anew. It takes a while for the conclusions of senior technical management to percolate to the level of senior

corporate management, but to the surprise of few the project was finally (and properly) terminated. The writeoff was reckoned in the hundreds of millions.

The essential precept of this book is that the issues attending the quality of software are those that lie at the heart of risk, cost, and schedule containment. Oh yes, and these issues have also to do with producing software that people are pleased with. After all, people ought to get something for their millions.

1.1 THE SIZE OF THE SOFTWARE PROBLEM

Millions? Try billions, about $40 billion for the United States alone. The cost of data processing has been quadrupling every ten years or so. Initially, hardware represented most of the cost, but few of us are old enough to remember those days. The cost of computer hardware performance has been decreasing by orders of magnitude each decade, sowing the field for ever larger, ever more ambitious software projects. To take a small-scale example, at one time I had a personal computer furnished with 256K of internal memory and two floppy disk drives, each capable of handling 360 kilobytes of data. For little more than this system had cost 18 months earlier, I replaced it with a new desktop with 640 kilobytes of internal memory and a processing speed roughly four times as fast. Also, instead of two floppy drives, the new system had one floppy and a 20 megabyte hard disk. Within two years, I had added a math coprocessor to increase speed and graphics to use newly available software. More telling, I was spending unproductive time at the tasks of conserving both RAM and disk space.

As cheaper hardware creates an insatiable demand for more software, we become more aware of the cost of that software—as software productivity has not in the least kept pace with hardware cost/performance ratios. Indeed, despite the introduction of structured programming, structured design, design tools, modern programming languages, and all the other shibboleths by which software engineers recognize each other, the most optimistic rate at which programmer productivity is increasing is about 5% per year. Five percent is scarcely enough. Quoting an eminent computer scientist, "A conservative estimate indicates a tenfold increase in the demand for software each decade."[1] Looking at the most voracious consumer of software development effort, the DoD, it is estimated that software costs will account for 10% of the defense budget by 1990.*

Cost is not the only concern. Where will all the programmers who are going to generate the new software come from? In the United States, we now have a half million or so, depending on the labor classifications one wants to include under the word "programmer." Without closing the gap between supply and demand, the number of people entering the software business is monotonically increasing. As reported in the *New York Times* March 23, 1986, basic data of the U.S. Department of Labor forecasts that the number of computer programmers will, by 1995, increase by 72% to a total of 586,000. During the same interval, the category the department calls "computer-analysts, data processors" is forecast to increase by 69% to a total of 520,000. Should we be looking forward to the day when computers consume the entire work force? Surely, even by the year 2020, we shall continue to need workers to grow food, build houses, and play Beethoven quartets. From every point of view, programmer productivity is a burning issue.

* Attributed to the Electronics Industry Association in a DoD briefing on the Strategy for a DoD Software Initiative (STARS).

Even as we have learned to look to Japan for workable approaches to improving the quality of our products, it is instructive to compare U.S. programming productivity with that of Japan. Although no two studies of productivity of either the United States or Japan arrive at the same numbers, all seem to find Japan far ahead of the United States. One citation suggests the average Japanese programmer produces 2,000 lines of code per month, while his or her American counterpart is generating fewer than 300.[2] I suspect that part of the reason for this discrepancy is that the average Japanese programmer is engaged in less complex projects than the average American programmer, but a more significant explanation for the difference probably lies with the greater use of software tools by the Japanese, and with the active participation of Japanese quality engineers.

Apart from the costs attending the development of new software, we are starting to lose sleep over the conceptual size of the projects now being considered. Where a large software system once numbered 100,000 lines of code, new systems are breaking the one million mark. With the power offered by modern chip technology, we can conceive of applications heretofore undreamed of. With the possible exception of science fiction writers, would anyone two decades ago have thought of the graphical computer-aided design systems that are used to generate complete VLSI (very large scale integration) chip designs? Would anyone have thought of connecting hundreds of such chips, once designed, into a distributed computing system capable of steering data and voice messages through various communications protocols across any path served by any combination of wire, fibre-optics, or satellite? Would anyone have conceived of real-time video picture enhancement capable of detecting and automatically pinpointing features of interest?

With these marvels—marvels even to those in the business of software—we encounter disquietingly complex conceptual structures wrought of our own cognitive processes. We have seen greater complexity before: the ecology of a farm pond, the structure and dynamics of a galaxy, our own bodies. But it is one thing to study that which exists, which has been designed or has evolved or has simply happened before we got there, and quite another thing to invent a structure of a complexity so great that it befuddles its own inventors. In a much-remarked paper,[3] David Parnas has questioned the very feasibility of the software that would be required to implement the Strategic Defense Initiative (Star Wars), and in the process has raised doubts among computer scientists and software engineers everywhere about how much we know about managing software projects that dare to extend the current bounds of complexity.

Awareness of the size of the software problem is reaching the highest levels of industry and government. In an attempt to improve the technology with which software complexity is managed, we now have in the United States three new enterprises directed squarely at the software technology problem:

- Microelectronics and Computer Technology Corporation, a consortium of 20 American suppliers of computer hardware and software. (MCC also addresses human interfaces and hardware.)

- Software Productivity Consortium, a cooperative effort of 15 defense electronics and aerospace contractors.

- Software Engineering Institute, an outgrowth of the DoD's ambitious project named Software Technology for Adaptable and Reliable Systems (STARS). The Institute is also affiliated with Carnegie-Mellon Institute.

The United States has no monopoly on software research and technology transfer. Abroad, three initiatives have attracted considerable note:

- Japan's Software Technology Center, with government, university, and industry participation.
- European Strategic Program for Research and Development in Information Technology (Esprit), formed by the European Economic Community, with tasks farmed out to companies in member countries.
- Alvey, an independent software initiative of the United Kingdom, which is also heavily involved in Esprit.

The funding of these ventures is continually undergoing revision, so it is not possible to state accurately the cost of these multinational and multicompany efforts. However, the total cost will certainly exceed two billion dollars by 1990—all this in the interests of attaining new levels of software complexity.

Most of the time, of course, we work within established limits of complexity. We do so, however, with programming teams smaller and perhaps less expert than those that will be required to implement the Strategic Defense Initiative, teams for whom the project at hand represents challenge enough. It seems that nearly every programming project taxes the imagination of its staff and provides new opportunities for error, error in understanding exactly what it is that is needed, error in translating the requirements into design, error in managing the entire process. As computers continue to proliferate and increase in power even as they decrease in space and cost, we find new applications for them that at once expand our capacities for performance and create new headaches for management.

Viewed from afar, software management is the management of complexity. Any number of approaches have been invented to ameliorate the task of managing complexity. Of these, perhaps the least difficult to implement is the concept of the *quality program*. Throughout the pages of this book we see how quality programs become part of the software solution. However, the notion of a quality program raises a problem of its own: If we are to use quality as a mechanism for reducing the pain of software management, we need to understand just what we mean by "quality" as it applies to software.

1.2 PERCEPTIONS OF SOFTWARE QUALITY

The connotation of "quality" seems to depend on the context in which it is used. One speaks of automobile quality in terms of the tolerance permitted in the fit of body panels, as measured in hundredths of inches. Using a different sense of quality, we also say that a car upholstered in real leather has greater quality than one furnished with vinyl seats (as measured in units of class). So too, with software. Analogous to automobile upholstery, a relational data base system has greater quality than a "flat" data base system. It gives the owner (or user) more utility, greater convenience, or, perhaps, just a sense of well-being. With regard to the fit of body panels, software may not develop squeaks with the passage of time, but as it ages it can become difficult to modify. Both notions of quality fall within J. M. Juran's concept of quality: fitness for use.

Restricted to software products, fitness for use is certainly inclusive. By why restrict quality to products? If, in our view of quality, we can also include the *process* that

results in products that are fit for use, we can also address the problems of managing complexity. That is, to join in the attack on the fundamental software problems now facing us, software quality programs should also encompass the suitability of programming processes to the formidable tasks of producing complex software. In short, we can identify two sets of objectives for software quality programs:

- Software products.
- Software processes.

As we shall see, the distinction between the two sets is more apparent than real. Indeed, actions taken to satisfy one are required to fully satisfy the other.

Quality of the Software Product

Let us return to the business of defining quality, the issues of leather vs. vinyl on the one hand and manufacturing tolerances on the other. Just how do we perceive the fitness for use of the software product? Our perception of fitness depends on how we use it. Let us take a software-intensive central office telephone switch as an example.

The subscriber perceives the product's fitness in terms of reliability and availability. When the phone goes off hook, the subscriber expects to hear a dial tone, if not by the time the instrument is placed against his or her ear, then no more than a second or two later. Once having dialed, the subscriber expects to be connected to the called terminal. The only exceptions permitted are busy signals and announcements that the number is no longer a working number (or one of the many other discrete announcements with which telephone companies remind us of how fumble-fingered we are). If the subscriber forwards calls from office to club, he or she expects that no potential customers will be lost if the afternoon is given over to tennis. Call forwarding, like all subscriber features, must work unerringly as advertised. In brief, telephone subscribers define telephone switch quality as service dependability.

Quite different is the view of the people operating the central office. They, too, regard dependability as paramount, but they see it in different ways. To provide the subscriber with new service features, the operational staff must be able to install these features (update the software) without interrupting service. Thus, they want their dialogue with the switch to be clear and unambiguous. There must be no references to data files normally hidden from their view. If the installers forget which step of the installation process they are in, they expect to find out by querying the system. If they enter plainly inappropriate data (e.g., a trunk label where a file name was required), they expect to be admonished. Central office technicians want software that will limit the likelihood and magnitude of human error, whether the task at hand is extending service, trouble-shooting a hardware problem, or going about such routine maintenance tasks as data backup operations.

The bookkeeping staff of the telephone company is also a user of the switch software. Although the staff anticipates its own fallibility from time to time in forgetting to redress a posting error, it will not accept software failures that charge calls to the wrong party. The bookkeeping staff perceives the quality of the switch in terms of the number of billing complaints received from subscribers.

Taking one more class of user, we have the maintenance programmers. Among the attributes of quality to which they are attuned, we find accurate and understandable

software documentation, readable and well-annotated source code, and a software structure that does not violate the specifications of the software tools at hand. For example, if a patch is necessary, the patch installation tools must be capable of ensuring that the correct locations of memory and none others are overwritten.

We could go on with others who are touched by the quality of switch software, but the point is made: Many factors enter into the quality of large software products.

Quality of the Process

The most straightforward perception of a "quality" programming process has to do with the result of the process: The process is of high quality if the resulting product is perceived to be of high quality. Quality products are necessary to the definition of a quality process, but insufficient. Although good product quality can dependably result only from a good process, other views of quality need to be entertained, especially if quality programs are to be funded. Apart from the incontrovertible connection between process and product, the success of a software project is measured not only in terms of the product but of the events that attended its development. If the project manager had to give up every second weekend to replan the project as a result of a succession of mishaps that affected schedules, or if the project manager's boss had to dig deep into the firm's pockets to pay for cost overruns, or if the programmers had to spend most of their time drearily writing documentation (which will read drearily) instead of designing, we can hardly say that the process is of high quality.

To digress briefly, we may note that many firms have looked to the manager (or director) of quality assurance (or quality control or product assurance or what have you) to "take care of software quality matters." Where this has happened, we can suspect that senior management does not know much about software quality. Moreover, we can almost bet that, unless they have taken pains to school themselves in systems or software methodology, managers of quality assurance, with their background of viewing quality in terms of the product or service delivered to customers, will decide that the quality department should get involved in software testing. Although more thorough testing can be expected to benefit product quality, it is only part of a bigger solution to product problems, and it misses altogether the broader scope of software quality.

Good products are only one of management's objectives. The other objectives are meeting schedule, meeting cost, and manageability. A quality process addresses all of the objectives. General managers perceive process quality (although they may not know its name) by the extent of customer satisfaction or increased sales or some similar objective measure, and by increased profit—the result of lower costs. The knowledgeable software manager perceives quality in the smoothness of the development or maintenance process: infrequent replanning exercises, awareness of schedule slippages in time to take remedial action, no embarrassing interviews with senior management to obtain more funds, and confidence that critical parts of the system have been identified and are competently being dealt with.

Indirectly, management's ability to cohabitate with software is affected by the view of the process held by the people of the process. Designers who have to wait their turn to get to the word processor to document their work become noticeably (sometimes clamorously) impatient. Programmers who are unsure of exactly what it is they are supposed to program go about their work dispiritedly. Equally cheerless are testers who find that faults in the test support software cause system crashes several times a day.

What do these disgruntled workers do? They do what software people have been doing ever since computers were invented: They leave to take jobs elsewhere. Now, given the romantic allure of the life of the gypsy, migratory programmers will leave sooner or later in any case, but the problems of management are exacerbated when sooner, rather than later, is the rule. Worst of all, the more gifted software personnel, who are often the most stable, are the first to refuse to put up with unproductive process crotchets. In short, the programming staff's view of the quality of the software process is everyone's business.

The problem with processes is that they are seldom designed. They happen or, more euphemistically, they evolve. A banking house starts off with a small programming staff, possibly contract programmers working under the direction of a salaried supervisor, to build a modest management information system (MIS) for the timely reporting of the firm's equity and debt positions. A process appropriate to the small scale of the problem is more or less defined, as often as not by ad hoc procedures transmitted verbally. Other than language processors, tools are few. A decade later, when the firm is attempting to network an integrated set of MIS packages to its branch offices, the process by which the new software will be generated is the original process, repeatedly patched, with each modification a quick fix in reaction to the shortcomings observed in the course of the immediately preceding project.

To take another typical example, an instrument manufacturer uses a microprocessor in a new equipment design. The programmers are the engineers who thought to incorporate computing power as a way of providing unique features or of reducing hardware elsewhere. Working under engineering disciplines (not the worst thing they could do), they get the chip programmed, although with more effort than had been anticipated. Although the programming turned out to be a learning process, the product is successful. Some years later, the manufacturer is using a dozen microprocessors in its equipments (now graduated to "systems") and the programmers are dealing with concurrency, online downloading of reactive software into microprocessors, and other software matters standing at a considerable remove from engineering affairs. Indeed, in recent years the company has hired professional programmers with no engineering training, but the software development process has remained tied to what was appropriate to the design of electronic equipment.

The conscious design of software development and maintenance processes is one of the pillars supporting the fairly new discipline of software engineering. In fact, the term "software engineering" was originally coined by Prof. F. L. Bauer of the Munich Technical University in contradistinction to the prevalent practice of "software tinkering."* "Software engineering" seems to have a cachet about it that has induced any number of managers, either out of ignorance or wishful thinking, to call their programmers software engineers, quite independently of the approach to programming. In any case, the establishment of software processes based on thoughtful methodologies and supported by modern technology is the hallmark of successful programming shops. The six projects cited in Section 1.1, three in the United States and three abroad, start with the precepts of software engineering as a given.

To see how appalling software tinkering can be, consider two examples of mismanagement I have personally encountered. The first is from the engineering-cum-programming shop of an instrument manufacturer. The target computer was a ruggedized

* In 1967 at a meeting of a study group on computer science established by the NATO Science Committee.[4]

minicomputer destined for an airborne application. To the surprise of the programming manager, the machine came unbundled, and the cost of software—an assembler (yes, the program was to be written in assembly language), linker, and loader—had to be negotiated separately. The programming manager, outraged that the cost of the software would be $10,000, found a much cheaper cross-assembler and linker-loader that could be hosted by a machine located in a service bureau in a nearby city. However, in order to use the cross-assembler, some preprocessing of the source code was required. The preprocessing would be done on the firm's general purpose computer. This is the software testing scenario that finally developed: Source code was entered into the local computer, a tape was output, the tape was delivered to the service bureau, and a day later a load tape for the target machine was delivered so that a test could be run. Total turnaround time was three to four days, depending on the time of day the first tape was produced. Not including lost labor time, the total cost of developing test load tapes during the months before the project—hopelessly behind schedule—was abandoned was ten times that of the scorned $10,000.

The second example of an impossible process concerns a major financial institution that hired droves of programmers to develop an electronic funds transfer (EFT) system. Previously, it had contracted for custom software or for the modification of off-the-shelf systems. The EFT system represented the firm's initial venture into in-house software development. The programmers were immediately set to work designing and coding, and in short order the director of information systems (or whatever his title was) was able to report hundreds of new lines of code generated each week. Not long before testing was to start it become apparent that the programmers had never been given a clear idea of what the EFT system was supposed to accomplish. Subsystems were defined in general terms, and it was in those general terms that the new staff happily set itself to the task of programming. However, there were no testable specifications for the subsystems and of course there was no system specification. As it happened, some of the code was salvaged by the new director of information systems—it is likely that more was salvageable if one only knew what it did—and the system was finally completed at a cost overrun of about 300% of the initial estimate.

In both of these examples, the programmers were aware of the impending disasters long before senior management. However, business organizational hierarchies being what they are, there was no way to tip off the top brass that unpleasant surprises were in the offing. Moreover, as is so often the case, senior management had only the vaguest notion of what software development was all about, and were not inclined to ask questions likely to yield recondite answers. (Even today, when all else fails, programming managers turn to obfuscation.) In the absence of a standard process founded on proven principles—even if proven elsewhere—management had no way to interpret project status reports and no way to demand remedial action (which in both cases included firing the responsible manager) while such action could still prevent disaster.

In stark contrast to these two examples of processes of abysmal quality, we can also find established processes based on the tenets of software engineering, as documented in countless journal articles and reports, conferences, workshops, symposia, and books. In the better programming shops, the processes continue to evolve, not to correct last year's disaster, but (often with quantified knowledge of the effects of the current process) to take advantage of the latest technology. These quality processes exist not by chance but as the result of a deliberate quality program.

1.3 ELEMENTS OF QUALITY

No matter how it is perceived, it is hard to find anyone who has not a good word to say for quality. We're all for it. The only question is how do we get it? Simply paying more is not the answer. We have paid plenty and failed to get quality. Indeed, overpaying for software is a symptom of poor process quality. No, money is not the source of software quality, although we shall see where it helps. In fact, there are not one, but three sources of software quality:

- **People.**
- **Technology.**
- **Management.**

The canvas painted by people, technology, and management does not leave out much. Still, given the universal pursuit of simple solutions to difficult problems, the breadth of the foundation of software quality is worth noting, even at the risk of banality.

Certainly, to say that people are a prerequisite for quality is to drop a bromide square in the middle of the floor. Nevertheless, quality requires qualified personnel, and personnel qualifications for software development include smarts, aptitude, education, training, and attitude. Each of these can be addressed by a quality program, although within fairly narrow limits for the first two. Section 1.1 spoke of the burgeoning size of the software work force, which does nothing for one's confidence in being able to recruit at some selected level of intelligence or aptitude. Finding the right educational background is also compromised in a sellers' market, but through tuition reimbursement and adjusted working hours, companies can take aggressive remedial action. Certainly, training and encouragement of positive attitudes lend themselves to management actions.

Apart from influencing the quality of the cognitive processes that produce software, we have also to deal with the problem of maintaining staff stability. If firms employing programmers have no monopoly on losing people at the worst possible times or losing people in whom considerable education and training have been invested, a new chapter in the annals of professional migration has nevertheless been written by the programming population. In some companies, annual turnover rates of 25% are considered normal. Understandably, staff stability is a concern of quality programs, and, as it happens, improving stability is often a concomitant of improvements in management and technology.

Of course, the interest in technology transcends the influence it has on dissuading people from leaving for more automated pastures. Quality derives directly from tools, defect removal techniques, the ease with which previously qualified software can be reused, the choice of programming languages, and the very choice of development methodologies. None of these factors is independent of the others, which requires software managers to have a working knowledge of the entire software engineering spectrum. The difficulty is compounded by the rate at which the technology is improving, that is, the rate at which programming is becoming software engineering. Looking about the world of programming, we see levels of technology (and corresponding quality) dating from 1970, from 1980, and from the mid-1980s. The remarkable thing is that we can see all technology epochs in the same firm. Many programming shops lend new meaning to the term "living history." For example, we might have part of the MIS department