The Software Quality Challenge

Watts S. Humphrey
The Software Engineering Institute

Many aspects of our lives are governed by large, complex systems with increasingly complex software, and the safety, security, and reliability of these systems has become a major concern. As the software in today's systems grows larger, it has more defects, and these defects adversely affect the safety, security, and reliability of the systems. This article explains why the common test-and-fix software quality strategy is no longer adequate, and characterizes the properties of the quality strategy we must pursue to solve the software quality problem in the future.

Today, many of the systems on which our lives and livelihoods depend are run by software. Whether we fly in airplanes, file taxes, or wear pacemakers, our safety and well being depend on software. With each system enhancement, the size and complexity of these systems increase, as does the likelihood of serious problems. Defects in video games, reservations systems, or accounting programs may be inconvenient, but software defects in aircraft, automobiles, air traffic control systems, nuclear power plants, and weapons systems can be dangerous.

Everyone depends on transportation networks, hospitals, medical devices, public utilities, and the international financial infrastructure. These systems are all run by increasingly complex and potentially defective software systems. Regardless of whether these large life-critical systems are newly developed or composed from modified legacy systems, to be safe or secure, they must have quality levels of very few defects per million parts.

Modern, large-scale systems typically have enormous requirements documents, large and complex designs, and millions of lines of software code. Uncorrected errors in any aspect of the design and development process generally result in defects in the operational systems. The defect levels of such operational systems are typically measured in defects per thousand lines of code. A one million line-of-code system with the typical quality level of one defect per 1,000 lines would have 1,000 undiscovered defects, while any reasonably safe system of this scale must have only a very few defects, certainly less than 10.

The Need for Quality Software

Before condemning programmers for doing sloppy work, it is appropriate to consider the quality levels of other types of printed media. A quick scan of most books, magazines, and newspapers will reveal at least one and generally more defects per page while even poor-quality software has much less than one defect per listing page. This means that the quality level of even poor-quality software is higher than that obtained for other kinds of human written text. Programming is an exacting business, and these professionals are doing extraordinarily high quality work. The only problem is that based on historical trends, future systems will be much larger and more complex than today, meaning that just to maintain today's defect levels, we must do much higher quality work in the future.

To appreciate the challenge of achieving 10 or fewer defects per million lines of code, consider what the source listing for such a program would look like. The listing for a 1,000-line program would fill 40 text pages; a million-line program would take 40,000 pages. Clearly, finding all but 10 defects in 40,000 pages of material is humanly impossible. However, we now have complex life-critical systems of this scale and will have much larger ones in the relatively near future. So we must do something, but what? That is the question addressed in this article.

Why Defective Systems Work

To understand the software quality problem, the first question we must answer is *If today's software is so defective, why aren't there more software quality disasters?* The answer is that software is an amazing technology. Once you test it and fix all of the problems found, that software will always work under the conditions for which it was tested. It will not wear out, rust, rot, or get tired. The reason there are not more software disasters is that testers have been able to exercise these systems in just about all of the ways they are typically used. So, to solve the software quality problem, all we must do is keep testing these systems in all of the ways they will be used. So what is the problem?

The problem is complexity. The more complex these systems become, the more different ways they can be used, and the more ways users can use them, the harder it is to test all of these conditions in advance. This was the logic behind the beta-testing strategy started at IBM with the OS/360 system more than 40 years ago. Early copies of the new system releases were sent to a small number of trusted users and IBM then fixed the problems they found before releasing the public version. This strategy was so successful that it has become widely used by almost all vendors of commercial software.

Unfortunately, however, the beta-testing strategy is not suitable for life-critical systems. The V-22 Osprey helicopter, for example, uses a tilting wing and rotor system in order to fly like an airplane and land like a helicopter. In one test flight, the hydraulic system failed just as the pilot was tilting the wing to land. While the aircraft had a built-in back-up system to handle such failures, the aircraft had not been tested under those precise conditions, and the defect in the back-up system's software had not been found. The defect caused the V-22 to become unstable and crash, killing all aboard.

The problem is that as systems become more complex, the number of possible ways to use these systems grows exponentially. The testing problem is further complicated by the fact that the way such systems are configured and the environments in which they are used also affect the way the software is executed. Table 1 lists some of the variations that must be considered in testing complex systems. An examination of the number of possibilities for even relatively simple systems shows why it is impractical to test all possibilities for any complex system. So why is complex software so defective?

Some Facts

Software is and must remain a human-produced product. While tools and techniques have been devised to automate the production of code once the requirements

and design are known, the requirements and design must be produced by people. Further, as systems become increasingly complex, their requirements and design grow increasingly complex. This complexity then leads to errors, and these errors result in defects in the requirements, design, and the operational code itself. Thus, even if the code could be automatically generated from the defective requirements and design, that code would reflect these requirements and design defects and, thus, still be defective.

When people design things, they make mistakes. The larger and more complex their designs, the more mistakes they are likely to make. From course data on thousands of experienced engineers learning the Personal Software Process℠ (PSP℠), it has been found that developers typically inject about 100 defects into every 1,000 lines of the code they write [1]. The distribution for the total defects injected by 810 experienced developers at the beginning of PSP training is shown by the total bars in Figure 1. While there is considerable variation and some engineers do higher-quality work, just about everybody injects defects.

Developers use various kinds of tools to generate program code from their designs, and they typically find and fix about half of their defects during this process. This means that about 50 defects per 1,000 lines of code remain at the start of initial testing. Again, the distribution of the defects found in initial testing is also shown by the test bars in Figure 1.

Developers generally test their programs until they run without obvious failures. Then they submit these programs to systems integration and testing where they are combined with other similar programs into larger and larger sub-systems and systems for progressively larger-scale testing. The defect content of programs entering systems testing typically ranges between 10 and 20 defects per 1,000 lines.

The most disquieting fact is that testing can only find a fraction of the defects in a program. That is, the more defects a program contains at test entry, the more it is likely to have at test completion. The reason for this is the point previously made about extensive testing. Clearly, if defects are randomly sprinkled throughout a large and complex software system, some of them will be in the most rarely used parts of the system and others will be in those parts that are only exercised under failure conditions. Unfortunately, these rarely used parts are the ones most likely to be exercised when such systems are subjected to the stresses of high transaction volume, accidents, failures, or military combat.

1.	Data rates
2.	Data values
3.	Data errors
4.	Configuration variations
5.	Number, type, and timing of simultaneous processes
6.	Hardware failures
7.	Network failures
8.	Operator errors
9.	Version changes
10.	Power variations

Table 1: *Some of the Possible Testing Variations*

Switches	Paths
1	2
4	6
9	20
16	70
36	924
49	3,432
64	12,870
81	48,620
100	184,756
400	1.38E+11

Table 2: *Possible Paths Through a Network*

The Defect Removal Problem

A defect is an incorrect or faulty construction in a product. For software, defects generally result from mistakes that the designers or developers make as they produce their products. Examples are oversights, misunderstandings, and typos. Furthermore, since defects result from mistakes, they are not logical. As a consequence, there is no logical or deductive process that could possibly find all of the defects in a system. They could be anywhere, and the only way to find all of the defects with testing is to exhaustively test every path, function, or system condition.

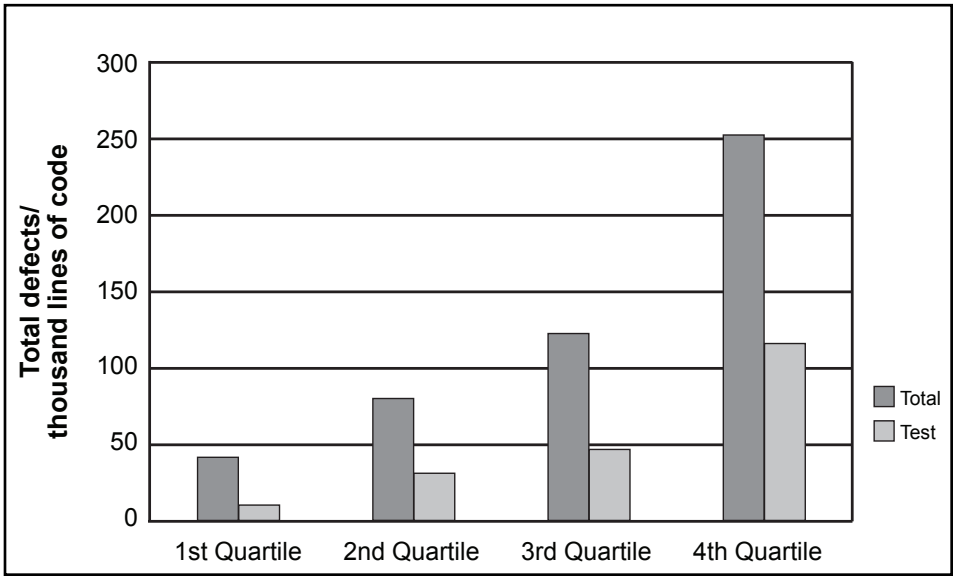
This leads to the next question which concerns the testing objective: "Must we find all of the defects, or couldn't we just find and fix those few that would be dangerous?" Obviously, we only need to fix the defects that would cause trouble, but there is no way to determine which defects these are without examining all of the defects. For example, a complex design defect that produced a confusing operator message could pose no danger while a trivial typographical mistake that changed a no to a yes could be very dangerous. Since there is no way to tell in advance which defects would be damaging, we must try to find them all. Then, after finding them, we must fix at least all of the ones that would be damaging.

The Testing Problem

Since defects could be anywhere in a large software system, the only way testing could find them all would be to completely test every segment of code in the entire program. To understand this issue, consider the program structure in Figure 2. This code fragment has one branch instruction at point B; three segments: A to B, B to C, and B to D; and two possible paths or routes through the fragment: A-B-C and A-B-D. So, for a program fragment like this, there could be defects on any of the code segments as well as in the branch instruction itself.

For a large program, the numbers of possible paths or routes through a program can vary by program type, but pro-

Figure 1: *Total and Test Defect Rates of 810 Experienced Engineers*



℠ Personal Software Process and PSP are service marks of Carnegie Mellon University.

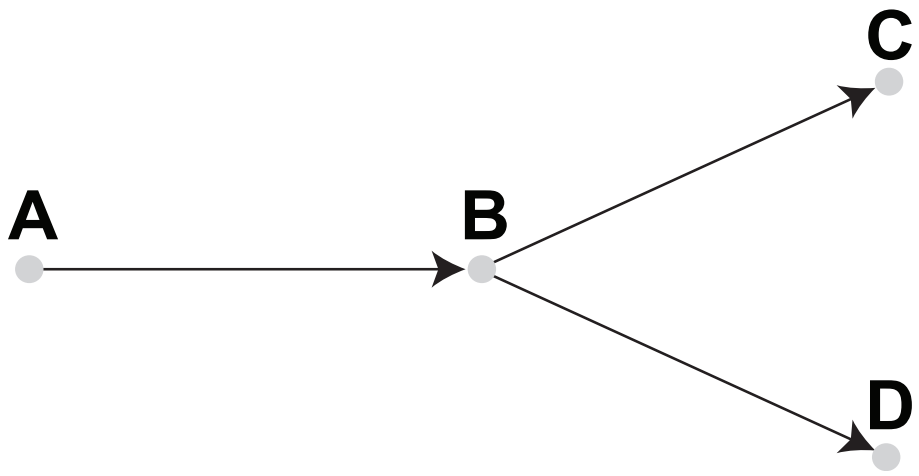


Figure 2: *A Three-Segment Code Fragment*

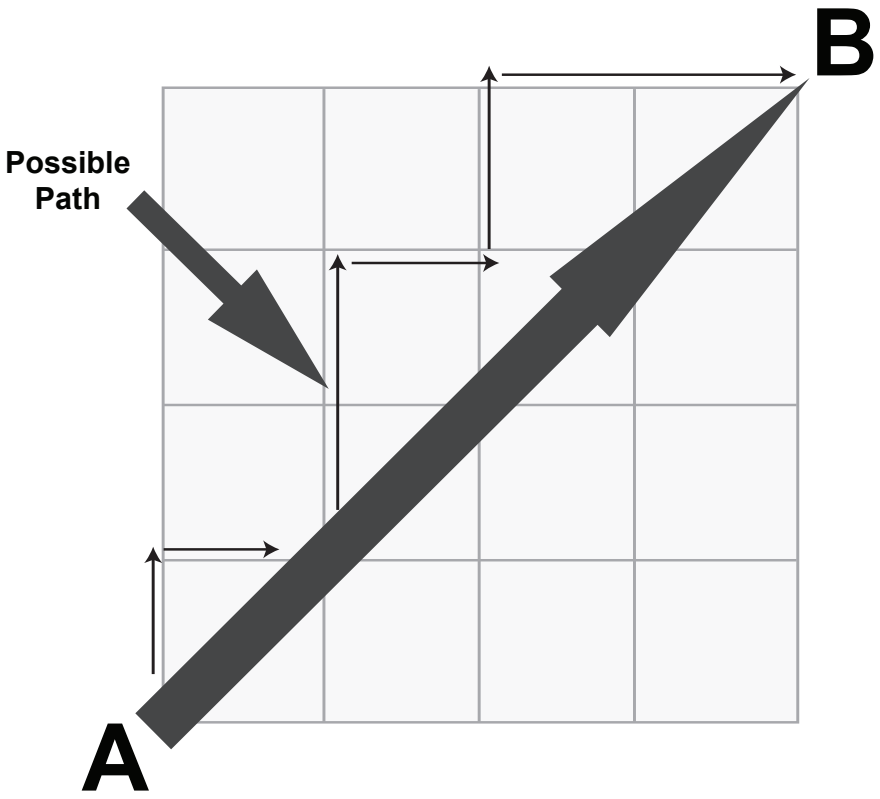
grams generally have about one branch instruction for every 10 or so lines of code. This means that a million-line program would typically have about 100,000 branch instructions. To determine the magnitude of the testing problem for such a system, we must determine the number of test paths through a network of 100,000 switches. Here, from Figure 3, we can calculate that, for a simple network of 16 switches, there are 70 possible paths from A to B. As shown in Table 2, the number of possible paths through larger networks grows rapidly with 100 switches having 184,756 possible paths and 400 switches having 1.38E+11 possible paths. Clearly, the number of possible paths through a system with 100,000 switches could, for practical purposes, be considered infinite. Furthermore, even if comprehensive path testing were possible, more than path testing would be required to uncover the defects that involved timing, synchronization, or unusual operational conditions.

Conclusions on Testing

At this point, several conclusions can be drawn. First, today's large-scale systems typically have many defects. Second, these defects generally do not cause problems as long as the systems are used in ways that have been tested. Third, because of the growing complexity of modern systems, it is impossible to test all of the ways in which such systems could be used. Fourth, when systems are stressed in unusual ways, their software is most likely to encounter undiscovered defects. Fifth, under these stressful conditions, these systems are least likely to operate correctly or reliably.

Therefore, with the current commonly used test-based software quality strategy, large-scale life-critical systems will be least reliable in emergencies – and that is when reliability is most important.

Figure 3: *Possible Paths Through a 16-Switch Network*



Successful Quality Strategies

Organizations have reached quality levels of a few defects per million parts, but these have been with manufacturing and not design or development processes. In the manufacturing context, the repetitive work is performed by machines, and the quality challenge is to consistently and properly follow all of the following steps:

- Establish quality policies, goals, and plans.
- Properly set up the machines.
- Keep the machines supplied with high-quality parts and materials.
- Maintain the entire process under continuous statistical control.
- Evaluate the machine outputs.
- Properly handle all deviations and problems.
- Suitably package, distribute, or otherwise handle the machine outputs.
- Consistently strive to improve all aspects of the production and evaluation processes.

While these eight steps suggest some approaches to consider for software development, they are not directly applicable for human-intensive work such as design and development. However, by considering an analogous approach with people instead of machines, we begin to see how to proceed.

The Eight Elements of Software Quality Management

The eight steps required to consistently produce quality software are based on the five basic principles of software quality shown in the Software Quality Principles sidebar. With these principles in mind, we can now define the eight steps required for an effective software quality initiative.

1. Establish quality policies, goals, and plans.
2. Properly train, coach, and support the developers and their teams.
3. Establish and maintain a requirements quality-management process.
4. Establish and maintain statistical control of the software engineering process.
5. Review, inspect, and evaluate all product artifacts.
6. Evaluate all defects for correction and

to identify, fix, and prevent other similar problems.
7. Establish and maintain a configuration management and change control system.
8. Continually improve the development process.

The following sections discuss each of these eight steps and relate them to the software quality principles as shown in the sidebar.

Step 1: Quality Policies, Goals, and Plans

Policies, goals, and plans go together and form the essential foundation for all effective quality programs. The fundamental policy that forms the foundation for the quality program is that quality is and must be *the* first priority. Many software developers, managers, and customers would argue that product function is critical and that project schedule and program cost are every bit as important as quality. In fact, they will argue that cost, schedule, and quality must be traded off.

The reason this is a policy issue is given in the first principle of software quality stated in the sidebar: *Properly managed quality programs reduce total program cost, increase business value and quality of delivered products, and shorten development times*. Customers must demand quality work from their suppliers and management must believe that if the quality program increases program costs or schedules, that quality program is not properly managed. There is, in fact, no cost/schedule/quality trade-off: manage quality properly, and cost and schedule improvements will follow. Everyone in the organization must understand and accept this point: it is always faster and cheaper to do the job right the first time than it is to waste time fixing defective products after they have been developed.

Once the basic quality policy is in place, customers, managers, and developers must then establish and agree on the quality goals for each project. The principal goal must be to find and remove all defects in the program at the earliest possible time, with the overall objective of removing all defects before the start of integration and system test. With the goals established, the development teams must make measurable quality plans that can be tracked and assessed to ensure that the project is producing quality work. This in turn requires that the quality of the work be measured at every step, and that the quality data be reviewed and assessed by the developers, their teams, management, and the customer. When defective work is found, it must be promptly fixed. The principle is that defects cost money. The longer they are left in the product, the more work will be built on this defective foundation, and the more it will cost to find and fix them [2].

Software Quality Principles*

1. Properly managed quality programs reduce total program cost, increase business value and quality of delivered products, and shorten development times.
   - 1.1. If cost or development times increase, the quality program is not being properly implemented.
   - 1.2. The size of a product, including periodic reevaluation of size as changes occur, must be estimated and tracked.
   - 1.3. Schedules, budgets, and quality commitments must be mutually consistent and based on sound historical data and estimating methods.
   - 1.4. The development approach must be consistent with the rate of change in requirements.
2. To get quality work, the customer must demand it.
   - 2.1. Attributes that define quality for a software product must be stated in measurable terms and formally agreed to between developers and customers as part of the contract. Any instance of deviation from a specified attribute is a *defect*.
   - 2.2. The contract shall specify the agreed upon quality level, stated in terms of the acceptable quantity or ratio of deviations (defects) in the delivered product.
3. The developers must feel personally responsible for the quality of the products they produce.
   - 3.1. The development teams must plan their own work and negotiate their commitments with management and the customer.
   - 3.2. Software managers must provide appropriate training for developers.
   - 3.3. A developer is anyone who produces a part of the product, be it a designer, documenter, coder, or systems designer.
4. For the proper management of software development projects, the development teams themselves must plan, measure, and control the work.
   - 4.1. Project teams must have knowledge and experience in the relevant technologies and applications domains commensurate with project size and other risk factors.
   - 4.2. Removal yield at each step and in total pre-delivery must be measured.
   - 4.3. Effort associated with each activity must be recorded.
   - 4.4. Defects discovered by each appraisal method must be recorded.
   - 4.5. Measurements must be recorded by those performing the activity and be analyzed by both developers and managers.
5. Software management must recognize and reward quality work.
   - 5.1. Projects must utilize a combination of appraisal methods sufficient to verify the agreed defect levels.
   - 5.2. Managers must use measures to ensure high quality and improve processes.
   - 5.3. Managers must use measurements with due respect for individuals.

* These principles were defined by a group of 13 software quality experts convened by Taz Daughtrey. The experts are: Carol Dekkers, Gary Gack, Tom Gilb, Watts Humphrey, Joe Jarzombek, Capers Jones, Stephen Kan, Herb Krasner, Gary McGraw, Patricia McQuaid, Mark Paulk, Colin Tully, and Jerry Weinberg.

Step 2: Train and Coach Developers and Teams

Quality work is not done by accident; it takes dedicated effort and properly skilled and motivated professionals. The third principle of software quality is absolutely essential: *The developers must feel personally responsible for the quality of the products they produce*. If they do not, they will not strive to produce quality results, and later trying to find and fix their defects will be costly, time consuming, and ineffective. Convincing developers that quality is their personal responsibility and teaching them the skills required to measure and manage the quality of their work, requires training. While it would be most desirable for them to get this skill and the required knowledge before they graduate from college, practicing software developers must generally learn them from using methods such as the PSP.

With properly trained developers, the development teams then need proper management, leadership, and coaching. Again, the Team Software Process℠ (TSP℠) can provide this guidance and support [3, 4, 5].

Step 3: Manage Requirements Quality

One fundamental truth of all quality programs is that you must start with a quality foundation to have any hope of producing a quality result. In software, requirements are the foundation for everything we do, so the quality of requirements is para-

℠ Team Software Process and TSP are service marks of Carnegie Mellon University.

mount. However, the requirements quality problem is complicated by two facts.

First, the quality measures must not be abstract characteristics of a requirements document; they should be precise and measurable items such as defect counts from requirements inspections or counts of requirements defects found in system test or customer use. However, to be most helpful, these quality measures must also address the precise understanding the developers themselves have of the requirements regardless of what the requirements originators believe or how good a requirements document has been produced. The developers will build what they believe the requirements say and not what the requirements developers intended to say. This means that the quality-management problem the requirements process must address is the transfer of understanding from the requirements experts to the software developers.

The second key requirements fact is that the requirements are dynamic. As people learn more about what the users need and what the developers can build, their views of what is needed will change. This fact enormously complicates the requirements-management problem. The reason is that people's understanding of their needs evolves gradually and often without any conscious appreciation of how much their views have changed. There is also a time lag: Even when the users know that their needs have changed, it takes time for them to truly understand their new ideas and to communicate them to the developers. Even after the developers understand the changes, they cannot just drop everything and switch to the new version.

To implement a change, the design and implementation implications of every requirements change must be appraised; plans, costs, and commitments adjusted; and agreement reached on how to incorporate this new understanding into the development work. This means that the requirements must be recognized as evolving through a sequence of versions while the development estimates, plans, and commitments are progressing through a similar but delayed sequence of versions. And finally, the product itself will ultimately be produced in a further delayed sequence of versions. The quality management problem concerns managing the quality and maintaining the synchronization of this sequence of parallel requirements, plan, design, and product versions.

Step 4: Statistical Process Control

While statistical process control is a large subject, we need only discuss two aspects: process management and continuous process improvement. The first aspect, process management, is discussed here, and process improvement is addressed in Step 8.

The first step in statistical process management is to redefine the quality management strategy. To achieve high levels of software quality, it is necessary to switch from looking for defects to managing the process. As noted earlier, to achieve a quality level of 10 defects per million lines with current software quality management methods, the developers would have to find and fix all but 10 of the 10,000 to 20,000 defects in a program with a 40,000 page listing. Unless someone devises a magic machine that could flawlessly identify every software defect, it would be clearly impossible to improve human search and analysis skills to this degree. Therefore, achieving these quality levels through better testing, reviews, or inspections is not feasible.

A more practical strategy is to measure and manage the quality of the process used to produce the program's parts. If, for example, we could devise a process that would consistently produce 1,000-line modules that each had less than a one percent chance of having a single defect, a system of 1,000 of these modules would likely have less than 10 defects per million lines. One obvious problem with this strategy concerns our ability to devise and properly use such a process.

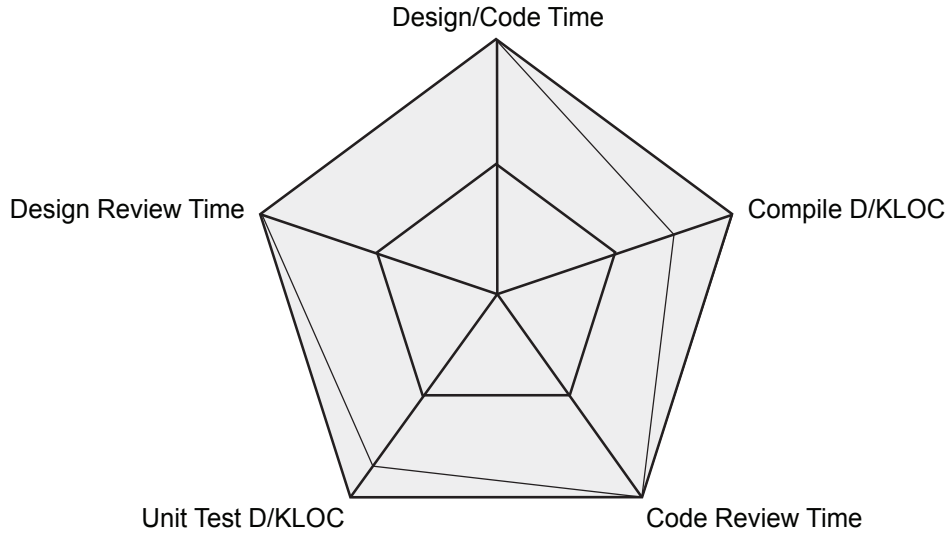
There has been considerable progress in producing and using such a process. This is accomplished by measuring each developer's process and producing a Process Quality Index (PQI). The TSP quality profile, which forms the basis for the PQI measure, is shown in Figure 5 [6]. Then, the developers and their teams use standard statistical process management techniques to manage the quality of all dimensions of the development work [7]. Data on early TSP teams show that by following this practice, quality is substantially improved [8].

Step 5: Quality Evaluation

Quality evaluation has two elements: evaluating the quality of the process used to produce each product element, and evaluating the quality of the products produced by that process. The reason to measure and evaluate process quality, of course, is to guide the process-improvement activities discussed in Step 8. The Capability Maturity Model® Integration (CMMI®) model and appraisal methods were developed to guide process-quality assessments, and the TSP process was developed to guide organizations in defining, using, and improving high-quality processes as well as in measuring, managing, and evaluating product quality.

To evaluate process quality, the developers and their teams must gather data on their work, and then evaluate these data against the goals they established in their quality plan. If any process or process step falls below the team-defined quality threshold, the resulting products must be evaluated for repair, redevelopment, or replacement, and the process must be brought into conformance. These actions must be taken for every process step and especially before releasing any products from development into testing. In product evaluation, the system integration and testing activities are also measured and evaluated to determine if the final product has reached a suitable quality level or if some remedial action is required.

Figure 5: *TSP Software Quality Profile* [6]



® Capability Maturity Model and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Step 6: Defect Analysis

Perhaps the most important single step in any quality management and improvement system concerns defect data. Every defect found after development, whether by final testing, the users, or any other means must be carefully evaluated and the evaluation results used to improve *both* the process and the product. The reason that these data are so important is that they concern the process failings. Every defect found after development represents a failure of the development process, and each such failure must be analyzed and the results used to make two kinds of improvements.

The first improvement – and the one that requires the most rapid turnaround time – is determining where in the product similar defects could have been made and taking immediate action to find and fix all of those defects. The second improvement activity is to analyze these defects to determine how to prevent similar defects from being injected in the future, and to devise a means to more promptly find and fix all such defects before final testing or release to the user.

Step 7: Configuration Management

For any large-scale development effort, configuration management (CM) is critical. This CM process must cover the product artifacts, the requirements, the design, and the development process. It is also essential to measure and manage the quality of the CM process itself. Since CM processes are relatively standard, however, they need not be discussed further.

Step 8: Process Improvement

The fundamental change required by this software quality-management strategy is to use the well-proven methods of statistical process control to guide continuous process improvement [7]. Here, however, we are not talking about improving the tolerances of machines or the purity of materials; we are talking about managing the quality levels of what people do, as well as the quality levels of their work products. While people will always make mistakes, they tend to make the same mistakes over and over. As a consequence, when developers have data on the defects they personally inject during their work and know how to use these data, they and their teammates can learn how to find just about all of the mistakes that they make. Then, in defining and improving the quality-management process, every developer must use these data to optimally utilize the full range of available defect detection and prevention methods.

Regardless of the quality management methods used (i.e., International Organization for Standardization, correctness-by-construction, or AS9100) continuous improvement strategies such as those defined by CMMI and TSP should be applied to the improvement process itself. This means that the process quality measures, the evaluation methods, and the decision thresholds must also be considered as important aspects of continuous process improvement. Furthermore, since every developer, team, project, and organization is different, it means that this continuous improvement process must involve every person on every development team and on every project in the organization.

Conclusion

While we face a major challenge in improving software quality, we also have substantial and growing quality needs. It should now be clear to just about everyone in the software business that the current testing-based quality strategy has reached a dead end. Software development groups have struggled for years to get quality improvements of 10 to 20 percent by trying different testing strategies and methods, by experimenting with improved testing tools, and by working harder.

The quality improvements required are vast, and such improvements cannot be achieved by merely bulling ahead with the test-based methods of the past. While the methods described in this article have not yet been fully proven for software, we now have a growing body of evidence that they will work – at least better than what we have been doing. What is more, this quality strategy uses the kinds of data-based methods that can guide long-term continuous improvement. In addition to improving quality, this strategy has also been shown to save time and money.

Finally, and most importantly, software quality is an issue that should concern everyone. Poor quality software now costs each of us time and money. In the immediate future, it is also likely to threaten our lives and livelihoods. Every one of us, whether a developer, a manager, or a user, must insist on quality work; it is the only way we will get the kind of software we all need.◆

Acknowledgements

My thanks to Bob Cannon, David Carrington, Tim Chick, Taz Daughtrey, Harry Levinson, Julia Mullaney, Bill Nichols, Bill Peterson, Alan Willett, and Carol Woody for reviewing this article and offering their helpful suggestions. I also much appreciate the constructive suggestions of the CROSSTALK editorial board.

References

1. Humphrey, W.S. PSP: A Self-Improvement Process for Software Engineers. Reading, MA: Addison-Wesley, 2005.
2. Jones, C. Software Quality: Analysis and Guidelines for Success. New York: International Thompson Computer Press, 1997.
3. Humphrey, W.S. Winning With Software: An Executive Strategy. Reading, MA: Addison-Wesley, 2002.
4. Humphrey, W.S. TSP: Leading a Development Team. Reading, MA: Addison-Wesley, 2006.
5. Humphrey, W.S. TSP: Coaching Development Teams Reading, MA: Addison-Wesley, 2006.
6. Humphrey, W.S. "Three Dimensions of Process Improvement, Part III: The Team Process" CROSSTALK Apr. 1998.
7. Florac, S., and A.D. Carleton. Measuring the Software Process: Statistical Process Control for Software Process Improvement. Reading, MA: Addison Wesley, 1999.
8. Davis, N., and J. Mullaney. "Team Software Process in Practice." SEI Technical Report CMU/SEI-2003-TR -014, Sept. 2003.

About the Author

Watts S. Humphrey joined the Software Engineering Institute (SEI) after his retirement from IBM. He established the SEI's Process Program and led development of the CMM for Software, the PSP, and the TSP. He managed IBM's commercial software development and was vice president of technical development. He is an SEI Fellow, an Association of Computing Machinery member, an Institute of Electrical and Electronics Engineers Fellow, and a past member of the Malcolm Baldrige National Quality Award Board of Examiners. In a recent White House ceremony, the President awarded him the National Medal of Technology. He holds graduate degrees in physics and business administration.

SEI
4500 Fifth AVE
Pittsburgh, PA 15213-2612
Phone: (412) 268-6379
Fax: (412) 268-5758
E-mail: watts@sei.cmu.edu