Acquiring Quality Software

Watts S. Humphrey
Software Engineering Institute

If you do not insist on getting quality software, you probably will not get it! That is the first principle of software quality. To get quality software at reasonable costs and on predictable schedules, you must follow the six principles of software quality. This article describes these principles and discusses how to apply them in software acquisition.

In today's software marketplace, the principal focus is on cost, schedule, and function; quality is lost in the noise. This is unfortunate since poor quality performance is the root cause of most software cost and schedule problems. However, as this article points out, there are proven ways to address this problem. The first step is adopting and demanding that vendors follow these six principles of software quality:

- **Quality Principle No. 1:** If a customer does not demand a quality product, he or she will probably not get one.
- **Quality Principle No. 2:** To consistently produce quality products, the developers must manage the quality of their work.
- **Quality Principle No. 3:** To manage product quality, the developers must measure quality.
- **Quality Principle No. 4:** The quality of a product is determined by the quality of the process used to develop it.
- **Quality Principle No. 5:** Since a test removes only a fraction of a product's defects, to get a quality product out of test you must put a quality product into test.
- **Quality Principle No. 6:** Quality products are only produced by motivated professionals who take pride in their work.

These are not just theoretical principles, and almost any software group can follow them, as demonstrated by the experiences of many organizations with the Software Engineering Institute's (SEI[SM]) Team Software Process[SM] (TSP[SM]). All it takes to start down this road is to recognize and act on quality principle No. 1.

Quality Principle No. 1

If the customer does not demand a quality product, he or she will probably not get one.

If you want quality products, you must demand them. But how do you do that? That is the subject of this article. I first define quality, then I discuss quality management, and then third, I cover quality measurement. Next I describe the methods for verifying the quality of software products before you get them, and finally, I give some pointers for those acquisition managers who would like to consider using these methods. That, of course, is the most critical point; even when you demand quality, if you cannot determine that you will get a quality product before you get it, you are no better off than you are today – struggling to recover from the effects of getting poor-quality products.

[SM] SEI is a service mark of Carnegie Mellon University.

> *"In the broadest sense, a quality product is one that is delivered on time, costs what it was committed to cost, and flawlessly performs all of its intended functions."*

Defining Quality

Product developers typically define a quality product as one that satisfies the customer. However, this definition is not of much help to you, the customer. What you need is a definition of quality to guide your acquisition process. To get this, you must define what quality means to you and how you would recognize a quality product if you got one.

In the broadest sense, a quality product is one that is delivered on time, costs what it was committed to cost, and flawlessly performs all of its intended functions. While the first two of these criteria are relatively easy to determine, the third is not. These first two criteria are part of the normal procurement process and typically receive the bulk of the customer's and supplier's attention during a procurement cycle, but the third is generally the source of most acquisition problems. This is because poor product quality is often the reason for a software-intensive system's cost and schedule problems.

Think of it this way: If quality did not matter, you would have to accept whatever quality the supplier provided, and the cost and schedule would be largely determined by the supplier. In simplistic terms, the supplier's strategy would be to supply whatever quality level he felt would get the product accepted and paid for. In fact, even if you had contracted for a specific quality level, as long as you could not verify that quality level prior to delivery and acceptance testing, the supplier's optimum strategy would be to deliver whatever quality level it could get away with as long as it was paid.

Since, at least for software, most quality problems do not show up until well after the end of the normal acquisition cycle, you would be no better off than before. I do not mean to imply that this is how most suppliers behave, but merely that this would be its most economically attractive short-term strategy. In the long term, quality work has always proven to be most economically attractive.

Addressing the Quality Problem

In principle, there are only two ways to address the software quality problem. First, use a supplier that has a sufficiently good record of delivering quality products so you will be comfortable that the products he provides will be of high quality. Then, just leave the supplier alone to do the development work. The second choice would be to closely monitor the development process the supplier uses to be assured that the product being produced will be of the desired quality.

While the first approach would be ideal, and that is the principle behind the successful Capability Maturity Model® Integration evaluation strategy, it is not useful when the supplier has historically had quality problems or where his current performance causes concern. In these

cases, you are left with the second choice: to monitor the development work. To do this, you must consider the second principle of quality management.

Quality Principle No. 2

To produce quality products consistently, developers must manage the quality of their work.

Managing Product Quality

While you may want a quality product, if you have no way to determine the product's quality until after you get it, you will not be able to pressure the supplier to do quality work until it is too late. The best time to influence the product's quality is early in its development cycle where you can determine the quality of the product before it is delivered and influence the way the work is done. At least you can do this if your contract provides you the needed leverage.

This, of course, means that you must anticipate the product's quality before it is delivered, and you must also know what to tell the supplier to do to assure that the delivered product will actually be of high quality. Therefore, the first need is to predict the product's quality before it is built. This is essential, for if you only measure the product's quality after it has been built, it is too late to do anything but fix its defects. This results in a defective product with patches for the known defects. Unless you have an extraordinarily effective test and evaluation system, you will not then know about most of the product's defects before you accept the product and pay the supplier.

While you might still have warranties and other contract provisions to help you recover damages, and you might still be able to require the supplier to fix the product's defects, these contractual provisions cannot protect you from getting a poor quality product. Because most suppliers are adept at avoiding liability for defects, you have not gained very much by contracting for quality. To get the benefits of including quality provisions in your contracts, you must determine the likely quality of the product during development.

Identifying Quality Work

To determine the likely quality of a product while it is being developed, we must consider the third principle of quality work.

Quality Principle No. 3

To manage product quality, the developers must measure quality.

To monitor product quality before delivery you must measure quality during development. Further, you must require that the developers gather quality measurements and supply them to you while they do the development work. What measures do you want, and how would you use them? This article suggests a proven set of quality measures, but first, to define these measures, we must consider what a quality product looks like.

While software experts debate this point, every other field of engineering agrees on one basic characteristic of quality: A quality product contains few, if any, defects. In fact, the SEI has shown that this definition is equally true for software. We also know that software professionals who consistently produce defect-free or near defect-free products are proud of their work and that they strive to remove all the product's defects before they begin testing. Low defect content is one of the principal criteria the SEI uses for identifying the quality of software products.

Defining Process Quality

To define the needed quality measures, we must consider the fourth quality principle.

Quality Principle No. 4

The quality of a product is determined by the quality of the process used to develop it.

This implies that to manage product quality, we must manage the quality of the process used to develop that product. If a quality product has few if any defects, that means that a quality process must produce products having few if any defects. What kind of process would consistently produce products with few if any defects? Some argue that extensive testing is the only way to produce quality software, and others believe that extensive reviews and inspections are the answer. No single defect-removal method can be relied upon to produce high-quality software products. A high-quality process must use a broad spectrum of quality management methods. Examples are many kinds of testing, team inspections, personal design and code reviews, design analysis, defect tracking and analysis, and defect prevention.

One indicator of the quality of a process is the completeness of the defect management methods it employs. However, because the methods could be applied with varying effectiveness, a simple listing of the methods is not sufficient. So, given two processes that use similar defect-removal methods, how could you tell which one would produce the highest quality products? To determine this, you must determine how well these defect-removal methods were applied. That takes measurement and analysis.

The Filter View of Defect-Removal

This leads us to the next quality principle.

Quality Principle No. 5

Since a test removes only a fraction of a product's defects, to get a quality product out of test, you must put a quality product into test.

This principle also applies to every defect-removal method, from reviews and inspections, through all the tests and other quality verification methods. Every defect-removal method only removes a fraction of the defects in the product; so to understand the quality of a development process, you must understand the effectiveness of the defect-removal methods that were used. Further, to predict the quality of the delivered product, you must measure the effectiveness of every defect-removal step.

This also means that the highest quality development process would be the one that removed the highest percentage of the product's defects early in the process and then had the lowest number of defects in final testing. Finally, this means that the highest-quality products are those with the fewest defects on entry into the final stage of testing.

Criteria for a Quality Process

To evaluate a process, you must measure that process and then compare the measures with your criteria for a quality process. This means that you must have criteria that define what a quality process looks like. From the filter view of defect removal shown in Figure 1, we see that defect removal is like removing impurities from water [1]. To get water that is pure enough to drink, we should find progressively fewer impurities in each successive filtration step. Finally, if we were going to actually drink the water ourselves, we would not want to find any impurities in the final filtration step.

In effect, this means that the last filtration step is really used to verify the quality of the water produced by the prior stages. If there were any significant impurities, you would want to put that water through the entire filtration process again, starting from the very beginning. Then you might be willing to take a drink. Similarly, for a software system, this suggests three quality criteria.

1. Most of the defects must be found early in the development process.

2. Toward the end of the process, fewer defects should be found in each successive filtration stage.
3. The number of defects found in the final process stages must be fewer than some predefined minimum.

Determining Process Quality

While these sound like appropriate process-quality criteria, they have one major failing – you will not have complete defect data until the end of the process after the product has been built, tested, accepted, and used. During the process you will only know the number of defects found so far and not the number to be found in future stages. This is a problem because a low number of defects in a defect-removal stage could be because the product was of high quality, because the defect-removal stage was improperly performed, or because the defect data on that stage were incomplete. This means that you must have multiple ways to determine the effectiveness of a defect-removal stage and that these ways must include at least one way to evaluate the effectiveness of that stage at the time that it is actually enacted. Partial defect data can be used to do that. In fact, without these data, there is no way to determine the effectiveness of the defect-removal stages.

The three things we can measure about a process stage are: (1) the time the developers spent in that stage, (2) the number of defects removed in that stage, and (3) the size of the product produced by that stage. Then, using historical data, you could compare the data for any type of defect removal stage with like data for similar stages from previously completed projects. As long as you had comparable data for completed projects, you could see what an effective review, inspection, or test looks like. You could then determine the quality of each stage of the current project and either agree that the supplier proceed or repeat some prior phases until the quality criteria were met.

In-Process Quality Measures

From data on 3,240 Personal Software Process℠ (PSP℠) exercise programs written by experienced software developers, the SEI has determined the characteristics of a high-quality software process [1]. These data are shown in Table 1, and they show that developers inject about 2.0 defects per hour during detailed design and find about 3.3 defects per hour during detail-level-design reviews (DLDR).

To find the defects injected in one hour of design work, the average developer would have to spend 60*2/3.3 = 36 minutes reviewing that design. Similarly, since developers inject an average of about 4.6 defects per hour during coding and find about 6.0 defects per hour in code reviews, this same average developer should spend about 60*4.6/6 = 46 minutes reviewing the code produced in each hour. Since there is considerable variation among developers, the SEI has established the general guideline that developers personally spend at least half as much time reviewing design or code quality as they spent producing that design or code.

Further, from data on many programs, we have found that, when there are fewer than 10 defects found while compiling each 1,000 lines of code and fewer than 5.0 defects found while unit testing each 1,000 lines of code, that program is likely to have few if any remaining defects [2]. Combining these criteria with an additional requirement that developers spend at least as much time designing a program as they spent coding it, gives the following five software process quality criteria [1].

Calculating the Quality Profile

The quality profile has five terms that are derived from the data shown in Table 1. The equations for these terms are as follows.

1. Design/Code Time = Minimum(design time/coding time: 1.0).
2. Design Review Time = Minimum(2* design review time/design time: 1.0).
3. Code Review Time = Minimum(2* code review time/coding time: 1.0).
4. Compile Defects/KLOC = Minimum (20/(10 + compile defects/KLOC): 1.0).
5. Unit Test Defects/KLOC = Minimum (10/(5 + unit test defects/KLOC): 1.0).

To derive the five profile terms, consider formula No. 3 for code reviews. According to Table 1, in one hour of coding, a typical software developer will inject 4.6 defects. Since this developer can find and fix defects at the rate of 6.0 per hour, he or she needs to spend 4.6/6.0 = 0.7667 of an hour, or about 46 minutes, reviewing the code produced in one hour. Since there is wide variation in these injection and removal rates, and since the number 0.7667 is hard to remember, the SEI uses 0.5 as the factor. Based on experience to date, this has proven to be suitable. Since these parameter values are sensitive to application type and operational criticality, we suggest that organizations periodically analyze their own data and adjust these values accordingly.

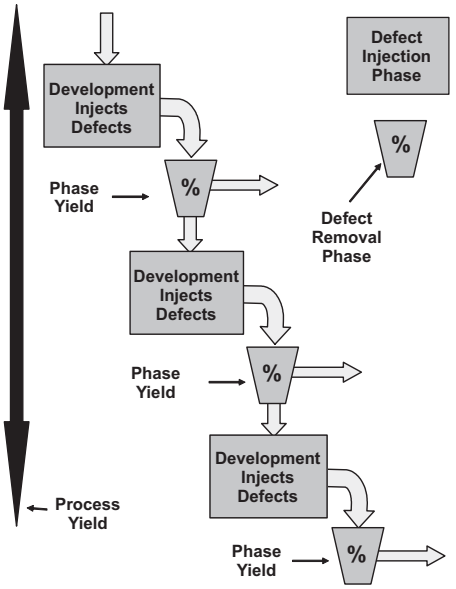

Figure 1: *The Defect-Removal Filtering Process*

The formula for the code review profile term compares the ratio of the actual time the developer spent reviewing code with the actual time spent in coding. If that ratio equals or is greater than 0.5, then the criteria are met. The factor of 2 in the equation is used to double both sides of this equation so it compares twice the ratio of review to coding time with 1.0. Also, to get a useful quality figure of merit, we need a measure that varies between 0 and 1.0, where 0 is very poor and 1.0 is good. Therefore, the equation's value should equal 1.0 whenever 2 times the code review time is equal to or greater than the coding time and be progressively less with lower reviewing times. This is the reason for the Minimum function in each equation, where Minimum(A:B) is the minimum of A and B. A little calculation will show that this is precisely the way equation No. 3 works. Equations No. 1 and No. 2 work in exactly the same way (except design time should equal or exceed coding time in equation No. 1).

To produce equations No. 4 and No. 5, the SEI used data it has gathered while training software developers for TSP

Table 1: *Defect Injection and Removal Rates (3,240 PSP Programs)*

Phase	Hours	Defects Injected	Defects Removed	Defects/Hour
Design	4,623.6	9,302		2.0
DLDR	1,452.7		4,824	3.3
Code	4,159.6	19,296		4.6
Code Review	1,780.4		10,758	6.0

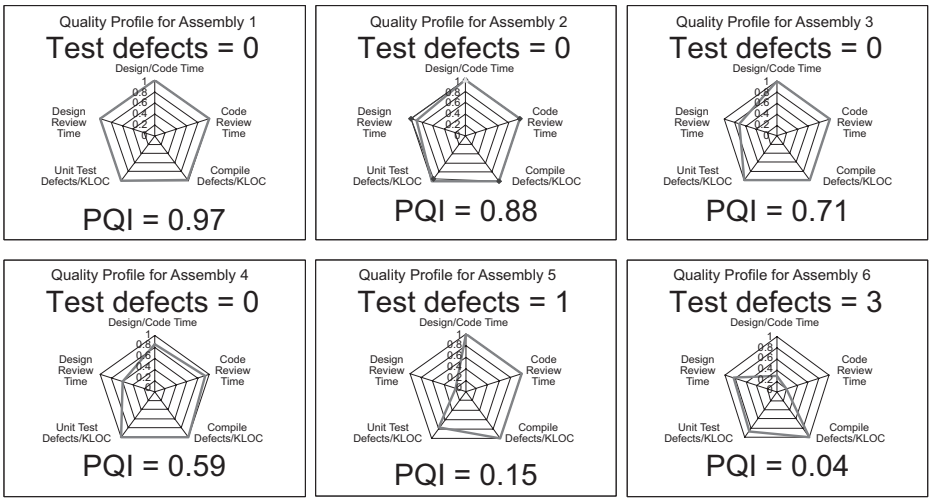

Figure 2: *Process Quality Profile (Six Programs)*

teams. It found that when more than about 10 defects/thousand lines of code (KLOC) were found in compiling, programs typically had poor code quality in testing, and when more than about five defects/KLOC were found in initial (or unit) testing, program quality was often poor in integration and system testing. Therefore, we seek an equation that will produce a value of 1.0 when fewer than 10 defects/KLOC are found in compiling, and we want this value to progressively decrease as more defects are found. A little calculation will show that this is precisely what equation No. 4 does. Equation No. 5 works the same way for the value of five defects/KLOC in unit testing.

One of the great advantages of these five criteria is that they can be determined at the time that process step is performed. Therefore, at the end of the design review for example, the developer can tell if he or she has met the design-review quality criteria. By plotting these five values on radar charts like those shown in Figure 2, it is relatively easy to identify a program's quality problems. The evaluation of these six profiles is as follows:

1. An excellent quality profile.
2. A similarly excellent quality profile.
3. A generally good quality profile with slightly too little design review time.
4. The design review measure is low, indicating potential problems that should be corrected with a repeated design review.
5. This product has a serious design review problem coupled with a unit testing problem. It should be reinspected. This product, when later tested had one defect found in final testing.
6. This product has serious design problems and an inadequate code review and should be replaced. This product had three defects found in subsequent testing.

Since these measures can all be available before integration and system test entry, and since they can be calculated for every component part of a large system, they provide the information needed to correct quality problems well before product delivery.

The Process Quality Index

For large products, it is customary to combine the data for all components into a composite system quality profile. Since the data for a few poor quality components could then be masked by the data for a large number of high quality components, it is important to have a way to identify any potentially defective system components. The process quality index (PQI) was devised for this purpose. It is calculated by multiplying together the five components of the quality profile to give a value between 0.0 and 1.0. Then the components with PQI values below some threshold can be quickly identified and reviewed to see which ones should be reinspected, reworked, or replaced.

Experience to date shows that, with PQI values above about 0.4, components typically have no defects found after development. Since the quality problems for large systems are normally caused by a relatively small number of defective components, the PQI measure permits acquisition groups to rapidly pinpoint the likely troublesome components and to require they be repaired or replaced prior to delivery. Once organizations have sufficient data, they should reexamine these criteria values and make appropriate adjustments.

Doing Quality Work

Since few software development groups currently gather the data required to use modern software quality management practices, we must consider the sixth principle of software quality.

Quality Principle No. 6

Quality products are only produced by motivated professionals who take pride in the quality of their work.

Because the measures required for quality management must be gathered by the software professionals themselves, these professionals must be motivated to gather and use the needed data. If they are not, they will either not gather the data or the data will not be very accurate. Experience shows that developers will only be motivated to gather and use data on their work if they use the data themselves, and if they believe that the practices required to consistently produce quality software products will help them do better work. Most developers who have used the TSP believe these things, but without proper training very few developers will.

While these measures and quality practices are not difficult, they represent a significant behavioral change for most practicing software professionals and their management. There are, however, a growing number of professionals who do practice these methods, and the SEI now has a program to transition these methods into general practice [1]. The methodology involved is the PSP, and to consistently use the PSP methods on a project, development groups must use the TSP. There is now considerable experience with these methods, and it shows that with proper use TSP teams typically produce defect-free or nearly defect-free products at or very close to their committed costs and schedules [2, 3, 4, 5].

Acquisition Pointers

Sound quality management is the key to software quality; without appropriate quality measures, it is impossible to manage the quality of a process or to predict the quality of the products that process produces. The developers must gather and analyze these data; they will not do this unless they know how to gather and how to use these data. This is why the sixth quality principle is critically important. Merely ordering the organization to provide the desired data will guarantee getting lots of numbers that are unlikely to be useful unless quality principle No. 6 is met. This requires motivating development management, and having development management train and motivate the developers in the needed quality measurement and management practices.

Once the developers regularly gather, analyze, and use these data, there only remains the question of how acquisition executives can get and use the data. This is both a contracting and a customer-supplier issue. Experience to date shows that when the developers use the TSP, you should have no trouble getting the required data [2, 3, 4, 5, 6, 7, 8].

The specific data needed to measure and manage software quality are the following:

1. The time spent in each phase of the development process. These times must be measured in minutes.
2. The number of defects found in each defect-removal phase of the process, including reviews, inspections, compiling, and testing.
3. The sizes of the products produced by each phase, typically in pages, database elements, or lines of code.

Planned and actual values are needed for these items, and these data should be for the smallest modules and components of the system. To establish and maintain the required management and developer motivation, these quality measurement and management requirements must be addressed both contractually and through management negotiation.

Conclusions

Poor quality performance damages a software development organization's cost and schedule performance and produces troublesome products. For acquirers to have a reasonable chance of changing the cost and schedule performance of their software vendors, they must demand effective quality management. The six principles of software quality reviewed in this article should help them do this.

By following these six principles and requiring suppliers to do so as well, you can consistently obtain quality software-intensive products at or very near to their committed costs and schedules.◆

References

1. Humphrey, Watts S. PSP: A Self-Improvement Process for Software Engineers. Reading, MA: Addison-Wesley, 2005.
2. Grojean, Carol A. "Microsoft's IT Organization Uses PSP/TSP to Achieve Engineering Excellence." CROSSTALK Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503Grojean.html>.
3. Davis, Noopur, and J. Mullaney. "Team Software Process (TSP) in Practice." Technical Report CMU/SEI-2003-TR-014. Pittsburgh, PA: Software Engineering Institute, Sept. 2003.
4. Humphrey, Watts S. Winning with Software: An Executive Strategy. Reading, MA: Addison-Wesley, 2002.
5. Humphrey, Watts S. TSP: Leading a Development Team. Reading, MA: Addison-Wesley, 2006.
6. Rickets, Chris A. "A TSP Software Maintenance Life Cycle." CROSSTALK Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503Rickets.html>.
7. Trechter, Ray, and Iraj Hirmanpour. "Experiences With the TSP Technology Insertion." CROSSTALK Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503Trechter.html>.
8. Tuma, David, and David Webb. "Personal Earned Value: Why Projects Using the Team Software Process Consistently Meet Schedule Commitments." CROSSTALK Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503Tuma.html>.

About the Author

Watts S. Humphrey joined the Software Engineering Institute (SEI) of Carnegie Mellon University after retiring from IBM in 1986. He established the SEI's Process Program and led development of the Software Capability Maturity Model®, the Personal Software Process℠, and the Team Software Process℠. During his 27 years with IBM, he managed all of IBM's commercial software development and was vice president of Technical Development. He is an SEI Fellow, an Association for 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. He has published several books and articles and holds five patents. In a White House ceremony, the president recently awarded him the National Medal of Technology. He has graduate degrees in physics and business administration.

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

Coming Events

January 4-7
Hawaii International Conference on System Sciences
Kauai, HI
www.hicss.hawaii.edu/HICSS39/apahome39.htm

January 8-10
IEEE Consumer Communications and Networking Conference
Las Vegas, NV
www.ieee-ccnc.org/ index.htm

January 8-11
Internet, Processing, Systems, and Interdisciplinaries (IPSI) USA 2006
Palo Alto, CA
www.internetconferences. net/california2006/index.html

January 11-13
The 33rd Annual Symposium on Principles of Programming Languages
Charleston, SC
www.cs.princeton.edu/~dpw/popl/06

February 6-9
Components for Military and Space Electronics Conference and Expo
Los Angeles, CA
www.cti-us.com/ucmsemain.htm

February 13-17
The Fifth International Conference on COTS-Based Software Systems
Orlando, FL
www.iccbss.org/2006

February 14-16
The International Association of Science and Technology for Development Conference on Software Engineering
Innsbruck, Austria
www.iasted.org/conferences/2006/Innsbruck/se.htm

May 1-4, 2006
2006 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org