A Personal Commitment to Software Quality[1]

Watts S. Humphrey
The Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA

Much of the following paper is an excerpt from one chapter of the book, A Discipline for Software Engineering (Addison Wesley) by the author. This textbook describes the personal software process (PSP) and provides a step-by-step program for its introduction. The book has been used as a text in six university graduate courses and is being used by several software organizations to help them introduce PSP methods. The Software Engineering Institute (SEI) is offering teach-the-teachers training courses for industrial groups interested in introducing the PSP to their organizations.

If you want a high quality software system, you must ensure each of its parts is of high quality. The personal software process (PSP) strategy focuses on managing the defects in the software you produce. By improving your defect management, you will produce more consistently reliable components. These components, in turn, can then be combined into progressively higher-quality systems.

While the quality benefits of this strategy are important, the productivity benefits are even more significant. Software productivity generally declines with increasing product size. [Boehm 81] One reason for this is the increased work entailed by the greater product volume. Another, more important reason is the quality of the product's parts. As products get larger, the increased amount of logic makes debugging much more difficult. More debugging in turn requires much more time in test.

When you produce parts of very high quality, your process will scale up with much less reduction in productivity. This is because as you add new elements to a progressively larger product, your testing need only concern the quality of the new parts. While you could have interface problems, the bulk of your testing will be localized. Hence you will largely retain your small program productivity when you develop larger programs.

[1]This work is supported by the U.S. Department of Defense.

To improve product quality, you must improve process quality. Doing this requires you to measure and track process quality. When you use the PSP, you will gather a lot of data that you can use to evaluate the quality of your processes. This paper relates process quality to product quality and shows how PSP data are used to measure and track process quality. It first defines software quality and then discusses the economic consequences of poor quality. It next deals with process measures, process benchmarking, PSP results, PSP status, and conclusions.

1. What Is Software Quality?

The principal focus of any software quality definition should be the users' needs. Crosby defines quality as "conformance to requirements." [Crosby] While one can debate the distinction between requirements, needs, and wants, quality definitions must consider the users' perspectives. The key questions then are: who are the users, what is important to them, and how do their priorities relate to the way you build, package, and support your products?

Product Quality

To answer these questions, you must recognize the hierarchical nature of software quality. First, a software product must provide functions of a type and at a time when the user needs them. If it does not, nothing else matters. Second, the product must work. If it has so many defects that it does not perform with reasonable consistency, the users will not use it regardless of its other attributes. This does not mean defects are always the highest priority, but they can be very important. If a minimum defect level has not been achieved, nothing else matters. Beyond this quality threshold, however, the relative importance of defects as well as of usability, compatibility, functionality, and all the other "ilities" depends on the user, the application, and the environment.

In a broad sense, the users' views of quality must deal with the product's ease of installation, operational efficiency, and convenience. Will it run on the intended system, will it run the planned applications, and will it handle the required files? Is the product convenient, can the users remember how to use it, and can they easily find out what they do not know? Is the product responsive, does it surprise the users, does it protect them from themselves, does it protect them from others, and does it insulate them from the system's operational mechanics? These and a host of similar questions are important to the users. While priorities will vary among users, quality has many layers, and no universal definition will apply in every case. If your software does not measure up in any single area that is important to your users, they will not judge your product to be of high quality.

While few software people will debate these points, their actions are not consistent with these priorities. Rather than devoting major parts of their development processes to installability, usability, and operational efficiency, they spend them on testing, the largest single cost element in most software organizations. Furthermore, these testing costs are almost exclusively devoted to finding and fixing defects.

When the quality of the parts of a software system is poor, the development process becomes fixated on finding and fixing defects. The magnitude of this fix process is often a surprise. As a result, the entire project becomes so preoccupied with defect repair that more important user concerns are ignored. When a project is struggling to fix defects in system test, it is usually in schedule trouble as well. The pressures to deliver become so intense that all other concerns are forgotten in the drive to fix the last defects. When the system tests finally run, everyone is so relieved that they ship the product. However, by fixing these critical system test defects, the product has reached only a bare minimum quality threshold. What has been done to assure the product is usable or installable? What about compatibility or performance? Has anyone checked that the documentation is understandable or that the design is suitable for future enhancement? Because the project's development team has been so fixated on fixing defects, it has not had the time or resources to address the issues that will ultimately be of greater concern to the users.

By sharply reducing the defect content of your small programs, use of the PSP will permit your projects to address the more important aspects of software quality. The quality of the product and the process thus go hand in hand. When a poor quality product is put into test, it generally means the development process could not be completed on schedule or within the committed costs. Conversely, a poor-quality process will generally produce a poor-quality product.

Even though software defects are only one facet of software quality, that is the quality focus of the PSP. It is not that controlling them should be the top priority but that effective defect management provides an essential foundation on which a truly comprehensive quality strategy can be built. While defects can come from many sources, with few exceptions software defects result from errors by individuals. Therefore, to properly address defects and the errors that cause them, we must deal with defects at the individual level. This is where defects are made, and this is where they should be found and fixed.

2. The Economics of Software Quality

Software quality can be viewed as an economic issue. You can always run another test or do another inspection. In large systems, every new test generally exposes a host of new defects. It is thus hard to know when to stop testing. While it is important to produce a quality product, each test costs money and takes time. Economics is thus an important quality issue not only because of this test decision but also because of the need to optimize life-cycle quality costs. The key to doing this is to recognize that you must put a quality product into test before you can expect to get one out. Section 3 of this paper shows why this is true.

The Costs of Finding and Fixing Defects

The economics of software quality largely concern the costs of defect detection,prevention, and removal. The cost of finding and fixing a defect includes the costs of each of the following elements:

- Determining that there is a problem

- Isolating the source of the problem

- Determining exactly what is wrong with the product

- Fixing the requirements as needed

- Fixing the design as needed

- Fixing the implementation as needed

- Inspecting the fix to ensure it is correct

- Testing the fix to ensure it fixes the identified problem

- Testing the fix to ensure it doesn't cause other problems

- Changing the documentation as needed to reflect the fix

While every fix will not involve every cost element, the longer the defect is in the product the larger the number of elements that will likely be involved. Finding a requirements problem in test can thus be very expensive. Finding a coding defect during a code review, however, will generally cost much less. Your objective thus should be to remove defects from the requirements, the designs, and the code as soon as possible. Reviewing and inspecting programs soon after they are produced minimizes the number of defects in the product at every stage. Doing this also minimizes the amount of rework and the rework costs. It will also likely reduce the costs of finding the defects in the first place.

Some Fix Time Data

There are not many published data on the time required to identify software defects. Following are some that are available:

- IBM: An unpublished IBM rule of thumb for the relative costs to identify software defects: during design, 1.5; prior to coding, 1; during coding, 1.5; prior to test, 10; during test, 60; in field use, 100.

- TRW: The relative times to identify defects: during requirements, 1; during design, 3 to 6; during coding, 10; in development test, 15 to 40; in acceptance test, 30 to 70; during operation, 40 to 1000. [Boehm 81]

- IBM: The relative time to identify defects: during design reviews, 1; during code inspections, 20; during machine test, 82. [Remus]

- JPL: Bush reports an average cost per defect: $90 to $120 in inspections and $10,000 in test. [Bush]

- Freedman and Weinberg: They report that projects that used reviews and inspections had a tenfold reduction in the numbers of defects found in test and a 50 percent to 80 percent reduction in test costs, including the costs of the reviews and inspections. [Freedman]

Some other defect data are shown in Table 1.

Clearly defect identification costs are highest during test and use. Thus anyone who seeks to reduce development costs or time should focus on preventing or removing defects before starting test. This conclusion is reinforced by the PSP data on the fix times for the 664 C++ defects and 1377 Pascal defects I found in the development of more than 70 small programs. These data show that fix times are 10 or more times longer during test and use than in the earlier phases. While this pattern varies somewhat between the two languages and by defect type, the principal factor determining defect fix time is the phase in which the defect was found.

A question often raised about these data is How do you know that the easy defects are not being found in the inspections with the difficult ones left for test? While this question cannot be resolved without substantially more statistical data, there is some evidence that inspections are as good as or better at finding the difficult-to-fix defects than is test.

- In the PSP, the pattern of fix times between reviews and test time is essentially the same regardless of defect type.

- Organizations that do inspections report substantial improvements in development productivity and schedule performance. [Dion, Humphrey 91]

- The PSP data show that reviews are two or more times as efficient as testing at finding and fixing defects. This is true of my own data, students' data, and working engineers.

- The fix advantage of reviews over tests is also true, almost regardless of the phase in which the defect was injected.

While the fix time can often be longer for design defects and much longer for requirements defects, the times to identify the defects appear to be the same. The reason for this appears to be that even trivial typographical defects can cause extraordinarily complex system behavior. Once these symptoms are deciphered, however, the fix is generally trivial. Conversely, very complex logic problems can have relatively obvious system consequences but be quite difficult to fix. It is also likely that the relative costs of finding and fixing sophisticated logic problems is a function of the application. I have seen data suggesting that defects in real-time systems or control programs can average as much as 40 hours each to find and fix in system test. These data are very important and you need to gather such data on your own work to determine the appropriate values for your environment.

The Economics of Defect Removal

From the data in Table 1, the times to find defects in test range from two to 20 hours. I have also seen numbers of 17 hours for an operating system project and 40 hours for a complex military system. The time to find defects in inspections, however, ranges from one quarter of an hour to one hour.

From the PSP data, it is also clear that even experienced software engineers normally inject 100 or more defects per KLOC into their programs. Some inject many more. While about half of these defects are typically found by the compiler, the rest must be found either by desk checking, inspections, or testing.

Using these data, you could estimate that a product of 50,000 LOC would enter test with about 50 or more defects per KLOC. This would mean that 2,500 or more defects must be found in test. For such a modest-sized product, five to 10 or more programmer hours would be required to find each defect. Hence testing could require 10,000 to 20,000 or more programmer hours. This is five to 10 person years. A five-person project working days, nights, and weekends might be able to finish in 18 months.

While there are data on the costs of finding defects with inspections and reviews, there is little data on the effectiveness of inspections and reviews. That is, if you were to inspect a software product that contained 100 defects, how many would you expect to find? We will call the percentage of the defects found the yield of the review or inspection. Again, there are no published data but my experience at IBM was that inspections typically yielded between 60 percent to 80 percent. Data from another organization support this with a reported 68% yield for one large operating system. With the PSP, combined design and code reviews can yield up to 80 percent. Typical individual code reviews, however, generally yield between 50 percent and 75 percent. As shown in Figure 1, 12 students who finished the 10 PSP exercises ended up with yields of between 50 percent and 100 percent with an average of about 70 percent.

Returning to the 50,000-LOC product example, assume you could find 70 percent of the defects by using inspections and reviews and your rates would be like those in Table 1. Inspections, at an average cost of 0.5 hours, would find 1750 defects and take 875 hours. The remaining 750 defects would have to be found in test at a cost of about 8 hours each. This whole process would take 6000 hours. While it would still take six to eight months of testing by five engineers, they would save a year.

You might ask why organizations do not do more reviews and inspections. There are two reasons why not. First, few organizations have the necessary data to make sound development plans. Their schedules are thus based largely on guesses, and these guesses are often unrealistic. When their plans are treated as accurate projections, the schedule pressure builds so quickly that all the engineers can do is react to the periodic crises. No one has time to think of anything but test, debug, and fix.

Second, yield is not generally managed. Without the discipline of a PSP, the engineers have no data on the number of defects they inject or the cost to find and fix those defects. They thus rarely appreciate the enormous costs that can be avoided by finding and fixing defects before test.

3. Process Measures

Juran describes the cost of quality measure as a way to "quantify the size of the quality problem in language that will have impact on upper management." [Juran] While your PSP costs will probably not be visible to your management, it is important that you begin to deal with quality as an economic issue. The cost of quality has three components: failure costs, appraisal costs, and prevention costs. [Crosby 83, Mandeville, Modarress] The definitions for these cost of quality components are the following:

- Failure costs: the costs of diagnosing a failure, making necessary repairs, and getting back into operation

.

- Appraisal costs: the costs of evaluating the product to determine its quality level

- Prevention costs: the costs associated with identifying the causes of the defects and the actions taken to prevent them in the future

For large projects, you should gather detailed data on these cost-of-quality components. For example, appraisal costs should include the costs of running test cases or of compiling when there are no defects. Similarly, the defect repair costs during inspections and reviews should be deducted from appraisal costs and counted as failure costs. For the PSP, we use a somewhat simpler definition as follows:

- Failure costs: the total spent in compile and test. Because the defect-free compile and test times are typically small compared to the defect-present times, they are included in failure costs.
- Appraisal costs: the times spent in design and code reviews plus any inspection times. Since the defect repair costs are generally a small part of review costs, the PSP leaves them in appraisal costs.
- If these numbers for inspection and review fix times or defect-free compile and test times were excessive, however, you could use the more-precise cost-of-quality definition.

Juran categorizes design reviews as prevention costs. For software, it is more appropriate to count both reviews and inspections as appraisal costs. The costs of most prototype development, causal analysis meetings, and process improvement action meetings should be classified as prevention costs. The PSP does not specifically include prevention actions because most defect-prevention work involves cross-project activities. To be most effective, process-improvement actions should be based on the experiences from several projects. Those improvements that are judged to be generally useful are incorporated in the organization's defined processes so that future projects can benefit from them.

Prototypes are typically developed to build a clear understanding of some requirement, function, or software structure. While there are many reasons for developing prototypes, they all stem from a desire to avoid making mistakes. For the PSP, developing prototypes can thus be considered defect-prevention actions.

Similarly, a formal specification of a software product is generally not required in order to design and build that product. It has been found, however, that formally defining a specification identifies unclear areas and often produces more complete and unambiguous specifications. Depending on your design practices, you could classify such work as either defect-prevention or project-performance costs.

The PSP Cost of Quality Measures

For the PSP, the cost-of-quality (COQ) measures are defined as follows:

Failure COQ=100*(compile time+test time)/(total development time)

Appraisal COQ=100*(design review time+code review time)/(total development time)

Total COQ=Appraisal COQ+Failure COQ

Appraisal as a % of Total Quality Costs=100*(Appraisal COQ)/(Total COQ)

A/FR ratio = Appraisal to failure cost ratio = (Appraisal COQ)/(Failure COQ)

The last two measures are useful for tracking process improvement or for comparing several processes. Figure 2 shows the trend of appraisal costs as a percentage of total quality costs for 12 students for the 10 PSP standard exercises. As you can see, appraisal costs generally increase throughout the PSP exercises.

From Figure 3 you can see that the numbers of test defects are much lower for the later programs. Thus there is a clear association between high appraisal costs and low test defects and a high association between high appraisal costs and improved product quality.

Yield Measures

A useful measure of process quality is total process yield, that is, the percentage of defects removed before the first compile or test. You can also apply this measure to every review, inspection, and test phase. Here, the yield of a phase is defined as follows:

Yield(step n)=100*(defects removed in step n)/(defects removed in step n+defects escaping step n)

Overall process yield is calculated as follows:

Yield(overall)=100*(all defects removed before compile)/(all defects in product at compile entry)

The concept of yield management is illustrated in Figure 4. Visualize the various development phases as producing a product that contains some number of defects. The various review, inspection, and test phases then act as filters to remove a percentage of these defects. Your strategy should thus be to

- reduce the number of defects you inject,
- improve the efficiency or yield of the filters,
- ensure the filters inject as few defects as possible, and
- compound the number of filter stages to achieve the desired product quality, cost, and schedule.

The cost of quality and the yield measures give you a balanced basis for understanding this strategy's costs and benefits.

4. Process Benchmarking

Various benchmarking techniques can be helpful in tracking processes and comparing them with similar processes used by other individuals, groups, or organizations. To use benchmarking, you first seek basic process measures that are independent of the process but that reflect its capability and robustness. If properly done, such comparisons will be valid for very different types of processes.

A useful, general-purpose process benchmark should do the following:

- Measure the ability of the process to produce high-quality products.
- Provide a clear ordering of process performance from best to worst.
- Indicate the ability of the process to withstand perturbations or disruptions.

- Provide required data that is objectively measurable.

- Provide data in a timely manner.

Benchmarking the Software Process

Unfortunately, no available software process measures meet all these criteria. While better measures may ultimately be developed, for now we must use the data we can get. We must thus devise measures that are as independent as we can make them. One useful approach is to use the combined measures of COQ and yield. For the ten PSP exercise programs, Figure 5 shows data for 12 students on a two-dimensional COQ/yield scale. While higher yields tend to be associated with a lower total cost of quality, the correlation is not strong. Figure 6, however, shows the overall COQ as a percentage of development did not fluctuate that much during the 10 programs, although there was a general convergence toward 30 percent.

Another way to look at cost of quality is to consider the ratio of appraisal costs to failure costs. This ratio would indicate the degree to which time is spent eliminating defects prior to the compile and test phases. As you can see from Figure 7, the A/FR, or appraisal-to-failure cost ratio, increases with the later programs. Figure 8 shows the relationship of process yield to the A/FR ratio. While there is considerable variation, high A/FR values are roughly associated with high yields. Figure 9 shows that the numbers of test defects declines quite sharply with increases in A/FR. A high A/FR value is clearly associated with low test defects and thus is a useful indicator of a quality software process.

Benchmarking Considerations

While the yield and the A/FR ratio appear useful for process benchmarking, they do not fully meet the criteria for a general-purpose benchmark because the COQ and yield measures are not easily standardized. Yield, as used here, is a measure of the fraction of the defects removed before compile and test. Such measures are highly sensitive to the definition of a defect and to the specific counting practices used. Few engineers, in fact, even count the defects they find in compile or unit test.

Similarly, the COQ measures depend on the process phase definitions. To compare your processes with others, you need comparable practices for counting defects and recording times. Variations in these practices could give quite different benchmark results for otherwise similar processes. Even with these problems, however, we do need measures. The yield and COQ measures are not perfect, but they can help you to assess changes in your process. As you track and assess your work, tracking the A/FR value, the yield, and their relationships over time will give you a sense of whether and how much your process is improving.

5. PSP Results

The results to date from the available data on four PSP courses show that improvement is substantial and almost universal. For example, the percentage improvement in the average number of defects per thousand lines of code (KLOC) from the beginning to the end of the 15-week PSP course shows improvements of from two to five or more times. The following table shows the percentage improvement in total defects, compile defects, and test defects for four courses.

Defect Types	**Class A**	**Class B**	**Class C**[2]	**Class D**
Total Defects	53.4%	45.8%	55.1%	80.1%
Compile Defects	68.8%	76.6%	75.7%	88.1%
Test Defects	68.8%	81.7%	64.2%	83.2%

In calculating these data, the average defects per KLOC for the first two programs was compared to the average for the last two. The numbers of engineers in classes A, B, C, and D were 4, 12, 6, and 19 respectively. The students in classes A, B, and C were moderately experienced engineers while most members of class D were graduate students with little industrial experience.

With these dramatic quality improvements, one might think that productivity would suffer. As the following table shows, however, productivity actually increased.

Average LOC/Hour	**Class A**	**Class B**	**Class C**	**Class D**
Exercises 1 and 2	19.9	31.4	11.4	13.8
Exercises 9 and 10	36.3	38.6	26.9	22.3
Percent Improvement	82.4%	22.9%	136.0%	61.6%

[2]Note that class C only did 9 of the exercises so the second set is the average of exercises 8 and 9.

While all groups improved, the amount of increase was inversely proportional to the initial productivity level. This implies that there is some limiting rate for lines of code (LOC) per hour. This is analogous to the 4:00 minute mile where the difference between world record holders and competent runners is only a few seconds. Group productivity rates appear to converge on about 40 LOC per hour but I have seen individual rates as high as 85 LOC per hour. For projects with 100 percent yield, however, the highest rate observed is 65 LOC per hour. While many factors will influence these rates and while some engineers will likely have higher rates for smaller or lower yield developments, the highest consistent rate for sustained high-quality work appears to fall somewhere around 70 LOC per hour. Substantially more data will be required, however, before such limits can be determined with any accuracy.

Assuming that there is such a rate limit, engineers could then make significant initial productivity improvements by defining and tracking their personal processes. Thereafter, productivity increases would generally not come from producing more LOC per hour. While better languages might provide some benefits, we will likely need improved architectural design concepts and more effective ways to reuse standard program elements. There are also significant improvement opportunities in the requirements and system design phases and in finding better ways to integrate small programs into larger systems. The PSP has not yet been used in these areas but its principles are applicable.

6. PSP Status

The original impetus for developing the PSP came from questions about the Software Engineering Institute's (SEI) capability maturity model (CMM). Many viewed the CMM as designed for large organizations and did not see how it could be applied to individual work or to small project teams. While the CMM does apply to both large and small organizations, more explicit guidance was clearly needed. The SEI thus started a process research project to examine ways individual engineers could apply level 5 process principles. After several years of research, means were devised to adapt 12 of the 18 CMM key process areas to the work of individual software engineers.

Experimental work was then started with several corporations to see how experienced engineers would react to the PSP and to explore introduction methods. It was found that experienced engineers are generally attracted by the PSP strategy and find the methods help them in their work. In the words of one engineer, "This isn't for the company, it's for me."

Six university courses have been taught to test the PSP course in software engineering curricula. The results were so positive that two universities have made the PSP a required course in software engineering. One has even made it the first course students must take in their software engineering masters degree program.

In developing the PSP industrial introduction strategy, we have found that software engineers have difficulty adopting new methods. They first learned to develop software during their formal educations and have since followed the same practices with a few adjustments and refinements. Since they are comfortable with these methods and have not seen compelling evidence that other methods work better, they are reluctant to try anything new. This problem is compounded by the fact that software engineers are rarely able to experiment. Everything they do is for delivery on a short and demanding schedule. An experiment would thus entail considerable risk. Not surprisingly, their reaction is to defer experimenting with new methods until they have some free time. Unfortunately, they never seem to have free time.

The current strategy is thus to introduce PSP methods in both industrial and academic environments with a formal course. In 15 weeks, the engineers develop ten small programming exercises and write five reports. They analyze their exercise data and see where and how the PSP methods help them to improve. The course is demanding, however, and active management support is required along with job time to complete the exercises.

7. Conclusions

One might ask why the software community has been so slow to adopt proven quality principles. The answer appears to be that these methods are difficult to introduce and are not intuitively obvious. Without convincing evidence, for example, few engineers believe it is more efficient to find defects by reviewing code than by testing and debugging. The PSP phased introduction strategy addresses this problem by demonstrating the methods to the engineers with their own data. By following a seven step process progression and completing 10 small programming exercises, engineers see how the PSP methods work for them.

By applying CMM principles to the work of individual software engineers, the PSP increases both the quality and the productivity of their work. While the PSP focuses on small example programs, engineers at Hewlett Packard, Digital Equipment Corporation, and the Advanced Information Services Corporation, have found that the PSP methods help them in doing their jobs.

Based on the experiences to date, universities should consider teaching the PSP methods and software organizations should review the PSP for possible introduction to their engineers.

References

[Ackerman] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski, "Software Inspections: An Effective Verification Process," *IEEE Software*, May 1989, pp 31-36.

[Boehm, 81] Barry W. Boehm. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Bush] Marilyn Bush, "Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory," 12th International Conference on Software Engineering, Nice, France, March 26-30, 1990, pp. 196-199.

[Crosby] Philip B. Crosby, *Quality is Free, The Art of Making quality Certain.* New York: Mentor, New American Library, 1979.

[Crosby 83] Philip B. Crosby, "Don't Be Defensive about the Cost of Quality," *Quality Progress*, April, 1983.

[Dion] Raymond Dion, "Process Improvement and the Corporate Balance Sheet," IEEE Software, July 1993, pp 28-35.

[Freedman] D. P. Freedman and G. M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluation Programs, Projects, and Products, Third Edition.* Little, Brown and Company, 1982.

[Humphrey 91] W.S. Humphrey, T.R. Snyder, and R. R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software*, July 1991, pp. 11-23.

[Juran] J. M. Juran and Frank M. Gryna, *Juran's Quality Control Handbook*, *Fourth Edition.* New York: McGraw-Hill Book Company, 1988.

[Mandeville] William A. Mandeville, "Software Costs of Quality," *IEEE Journal on Selected Areas in Communications,* Vol. 8, No. 2, February 1990.

[Modarress] Batoul Modarress and A. Ansari, "Two New Dimensions in the Cost of Quality," *International Journal of Quality and Reliability Management*, 4, 4.

[O'Neill] Don O'Neill, personal communication.

[Ragland] Bryce Ragland, "Inspections are Needed Now More than Ever," Journal of Defense Software Engineering #38, Published by the Software Technology Support Center, DoD, Nov. 1992.

[Remus] H. Remus and S. Ziles, "Prediction and Management of Program Quality," Proceedings of the Fourth International Conference on Software Engineering, Munich, Germany, 1979, pp341-350.

[Russell] Glen W. Russell, "Experience with Inspections in Ultralarge-Scale Developments," *IEEE Software,* Jan. 1991, pp 25-31.

[Shooman] M. L. Shooman and M. I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," Proc. 1975 Conference on Reliable Software, IEEE, NY, Catalog no. 75 CHO 940-7CSR, p347.

[vanGenuchten] Michael vanGenuchten, personal communication.

[Weller] E. F. Weller, "Lessons Learned from Two Years of Inspection Data," IEEE Software, September, 1993, pp 38-45.

Table 1 Hours to Find a Defect

Reference	Inspection	Test	Use
Ackerman	1	2-10	
O'Neil	.26		
Ragland		20	
Russel	1	2-4	33
Shooman	.6	3.05	
vanGenuchten	.25	8	
Weller	.7	6	

This article originally appeared in the December 1994 issue of Ed Yourdon's *American Programmer* journal, which focused on "Peopleware." Fore more information about *American Programmer,* contact:

Beth O'Neill
Cutter Information Corp
37 Broadway
Arlington, MA 02174-5552
U.S.A.

Phone: (617) 648-8702 or in North America 800 964-8702 toll free

FAX: (617) 648-1950 or in North America 800 888-1816 toll free

Internet: 74017.653@compuserve.com