8

Software Quality

Software quality is a question of scale. With traditional methods, software developers produce their code as best they can and then they review this code, hold team inspections, and run tests. Their objective is to find and fix as many of the program's defects as they can. Although this approach has traditionally produced programs that run and even run pretty well, it has not produced code that is of high enough quality or sufficiently safe and secure to use for critical applications.

Perhaps the best way to explain the quality problem is by examining some data. Table 8.1 shows the defect densities of the programs delivered by organizations at various CMM maturity levels (Davis and Mullaney 2003). The CMM, or Capability Maturity Model, was introduced by the Software Engineering Institute to help organizations assess the capability of their software groups. At CMM level 1, typical defect content is about 7.5 defects per thousand lines of code. By the time organizations get to CMM level 5, they have applied all of the traditional software quality methods to find and fix the program's defects. These methods generally include requirements inspections, design inspections, compiling, code inspections, and a lot of tests. Although the improvement from CMM1 to CMM5 is significant, today's large-scale software systems have millions of lines of code.

TABLE 8.1 DELIVERED DEFECT DENSITIES BY CMM LEVEL

CMM Level	Defects/KLOC	Defects/MLOC
1	7.5	7,500
2	6.24	6,240
3	4.73	4,730
4	2.28	2,280
5	1.05	1,050

Therefore, a delivered 1,000,000-LOC program produced by a CMM level-5 organization would likely have about 1,000 undiscovered defects.

The key question is, How many defects can a program have and still be considered a quality product? When a million-line program has thousands of defects, it is almost certainly not of high enough quality to be safe or secure. With 100 defects per million LOC, it would have much higher quality, but most of us would still probably not feel safe riding in an airplane or driving a car controlled by such software. At 10 defects per million LOC, we might begin to consider the point debatable, and at 1 defect per MLOC, most of us would agree that, although not completely satisfied, we would begin to think that program quality was approaching an acceptable level.

So what would it take to get 1 or 10 defects per million LOC? I counted one of my C++ programs and found that a 996-LOC listing had 30 pages. Assuming that the data for your programs were similar, a 1,000-LOC program would, on average, have about a 30-page listing. At one defect per KLOC, level-5 organizations are then getting quality levels of about one delivered defect in every 30 pages of source-program listings. Although one defect in 30 pages is not bad when compared with other human-produced products, it is still 1,000 defects per MLOC. To get 100 defects per MLOC, we must have only one defect left in 300 listing pages, and for 10 defects per MLOC, we could miss at most one defect in 3,000 pages of listings. I cannot imagine reviewing and inspecting 3,000 pages of listings and missing at most one single defect. As this seems like a preposterous challenge, what can we do?

The answer, of course, is that people cannot inspect 3,000 pages and be confident of missing no more than one single defect. At one defect in 30 pages, today's level-5 organizations are near the limit of what people can achieve with the present commonly used methods. To get better quality levels, we must change our approach. Instead of searching for defects, we must focus on the process. We must measure the likelihood of a 1,000-LOC module having defects. If we could devise a process that would assure us, at development time, that a 1,000-LOC component had less than a 1% chance of having a defect, our systems would have about 10 defects per MLOC.

The PSP has process quality measures that help you to do this. However, because many of the required quality actions involve both personal and team work, you need the help of teammates to produce the highest-quality software. Then, to further improve product quality, you must improve the quality of your process. This requires measuring and tracking your personal work. This chapter relates process quality to product quality and shows how to use the PSP data to measure, track, and improve process quality. After defining software quality, the chapter covers the economics of quality, personal quality practices, quality measurement and management, PSP improvement practices, and defect prevention. A section at the end of the chapter discusses ways to use PSP quality-management methods when enhancing large existing products. Although the chapter's focus is principally on personal practices, the methods described are even more effective when used by all the members of a development team.

8.1 The PSP Quality Strategy

To produce a high-quality software system, each of that system's parts must also be of high quality. The PSP strategy is to manage the defect content of all parts of a system. By doing this, you will produce consistently reliable components that can then be combined into high-quality systems. Though the quality benefits of this strategy are important, the productivity benefits are even more significant. Software productivity generally declines with increasing product size (Boehm 1981). The primary reason is that it takes longer to find and fix defects in a big product than in a small one. Therefore, as products get larger, the greater volume of code both increases the number of defects and makes each defect harder to find. The common result is a great deal more time spent in testing.

With very high-quality parts, the software development process scales up without losing productivity. As you add high-quality elements to a high-quality base product, testing need only address the new parts. Although there could be systemic problems, the bulk of the defects will be localized. Hence, with high quality, you can largely retain small program productivity when developing larger products.

8.2 What Is Software Quality?

The principal focus of any software quality definition should be on the users' needs. Crosby defines quality as "conformance to requirements" (Crosby 1979). The key questions therefore are, Who are the users, What is important to them, and How do their priorities relate to the way that you build, package, and support products?

Product Quality

To answer these questions, we must consider the hierarchical nature of software quality. First, 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 that defects are always the highest priority, but that they can be very important. If a minimum quality level is not achieved, nothing else matters. Beyond this quality threshold, the relative importance of performance, safety, security, usability, compatibility, functionality, and so forth depend on the user, the application, and the environment. However, if the software product does not provide the right functions when the user needs them, nothing else matters.

Although these user priorities are generally recognized as paramount, development organizations typically devote most of their time to finding and fixing defects. Because the magnitude of this fix process is often underestimated, most projects become preoccupied with defect repair and ignore the other more important user concerns. When a project team is struggling to fix defects in testing, it is usually late and the pressures to deliver are so intense that all other concerns are forgotten. Then, when the tests finally run, everyone is so relieved that they ship the product. However, by fixing these critical test defects, the product has reached only a minimum quality threshold. What has been done to make the product usable or installable? What about compatibility, performance, safety, or security? Has anyone checked the documentation or made the design suitable for future enhancement? Because of the excessive time required to fix defects, little or no time has been spent addressing the issues that are ultimately more important to the users.

With fewer test defects, projects can address the quality aspects that customers feel are most important. Product and process quality thus go hand in hand. A heavy reliance on testing is inefficient, time-consuming, and unpredictable. Then, when the development groups finally stop testing, they deliver a poor-quality product to the users. Though defects are only part of the quality story, they are the quality focus of the PSP. Defects are not the top priority, but effective defect management is essential to managing cost, the schedule, and all other aspects of product quality. Because defects result from errors by individuals, to effectively manage defects, you must manage personal behavior. You are the source of the defects in your products and you are the only person who can prevent them. You are also the best person to find and fix them.

8.3 The Economics of Software Quality

Software quality is an economic issue. You can always run another test or do another inspection. In most large systems, because every new test exposes new defects, it is hard to know when to stop testing. Although everyone agrees that

quality is important, each test costs money and takes time. The key fact in optimizing life-cycle quality costs is that the cost of finding and fixing defects increases sharply in later life-cycle phases. However, it doesn't just cost a little more; the difference is enormous. To see why these costs are so high, consider the steps involved in finding and fixing defects:

1. The user experiences a problem and determines that it is a software problem.
2. The user reports the problem to a software specialist.
3. The software specialist verifies that there is a software problem.
4. The specialists reports the source of the problem to a developer.
5. The developer determines what is wrong with the product.
6. The developer fixes the product.
7. A development group inspects the fix to ensure that it is correct.
8. A test group tests the fix to ensure that it fixes the problem.
9. The test group tests the fix to ensure that it doesn't cause other problems.
10. The developers change the documentation to reflect the fix.
11. The release group rebuilds the product with the fix included.
12. The release group distributes the rebuilt product to the users.
13. The user installs the rebuilt product.
14. The user confirms that the fix actually corrected the problem.

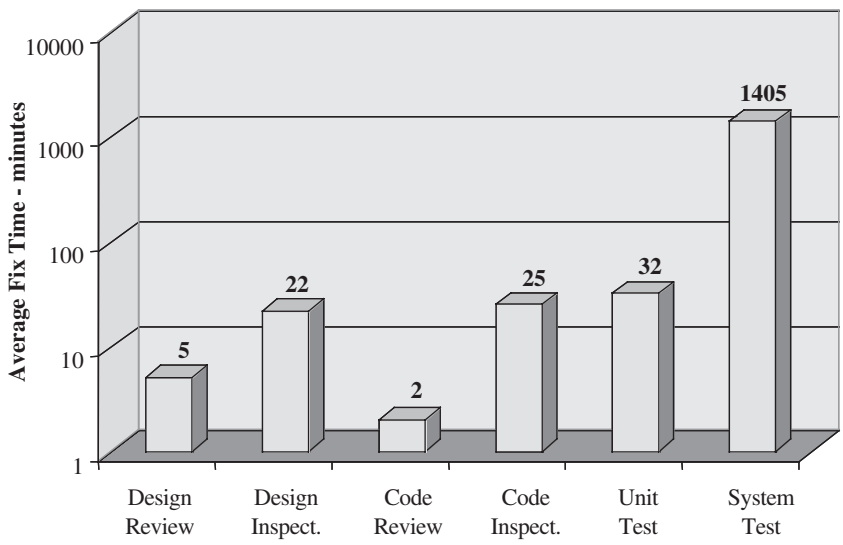
Although every fix may not involve every cost element, the longer the defect is in the product, the greater its impact. Finding a requirements problem during customer use can be very expensive. Finding a coding defect during a code review, however, will cost much less. The objective, therefore, must be to remove defects from the requirements, the designs, and the code as soon as possible. By reviewing and inspecting every work product as soon as you produce it, you can minimize the number of defects at every stage. This also minimizes the volume of rework and the rework costs. It also improves development productivity and predictability as well as accelerating development schedules.

Testing is expensive for two reasons. First, a test produces a symptom, and then someone must find and fix the defect. Though these find-and-fix times can be short for many defects, occasionally they can take many hours or even days. Second, testing is expensive because each test checks only one set of operating conditions. Even relatively simple systems produce many possible data values, operating parameters, error conditions, and configuration combinations. Because some defects will affect program behavior under only limited conditions, finding most of the defects in testing would require running a very large number of tests. Even then, you could only be confident that the system would run when it was used in exactly the way it was tested.

Although testing can be very effective at identifying performance, usability, and operational problems, it is not an effective way to remove volumes of defects. Think about it this way: if you knew that you had to do some repetitive task thousands of times, wouldn't it be a good idea to find out the fastest and most effective way to do it? A great deal of data is available on the time required to find and fix software defects. These data all agree that the later a defect is found, the more it costs to find and fix it. Table 8.2 shows a summary of published data from several sources, and Figure 8.1 shows defect-repair data for a Xerox TSP team. Figures 8.2 and 8.3 show PSP data on the fix times for the 664 C++ defects and 1,377 Pascal defects I found while developing 72 module-size programs during my early PSP work. Fix times are clearly much longer during testing and use than in the earlier phases. Although this pattern varies somewhat by language and defect type, the principal factor determining defect fix time is the phase in which the defect was found and fixed.

Figure 8.4 shows a histogram of the per-defect times for 810 developers for the final four exercise programs in a PSP class. The times reflect the average time required to find and fix a defect in design reviews, code reviews, and testing. These data are for 3,240 programs that had a total of 87,159 LOC and included a total of 28,251 defects.

Even during development, unit testing is much less effective at finding and fixing defects than either design reviews or code reviews. The testing histogram at



Source: Xerox

FIGURE 8.1 DEFECT-REPAIR TIME FOR A XEROX TSP TEAM

TABLE 8.2 COST OF FINDING AND FIXING DEFECTS

Reference	Inspection	Test	Use
IBM (Remus and Ziles 1979)	1	4.1 times inspection	
JPL (Bush 1990)	$90–$120	$10,000	
Ackerman et al. 1989	1 hour	2–20 hours	
O'Neill 1994	0.26 hour		
Ragland 1992		20 hours	
Russell 1991	1 hour	2–4 hours	33 hours
Shooman and Bolsky 1975	0.6 hour	3.05 hours	
vanGenuchten 1994	0.25 hour	8 hours	
Weller 1993	0.7 hour	6 hours	

the back of Figure 8.4 has many low rates, and the code review (CR) bar in the middle indicates many programs for which the developers found and fixed nearly 10 defects per hour or more. The zero-defect cases at the left were for programs that had no defects in that phase. The zero-defect design review bar (DLDR) is high because few developers find many defects in design reviews until they start measuring and managing the quality of their personal work.

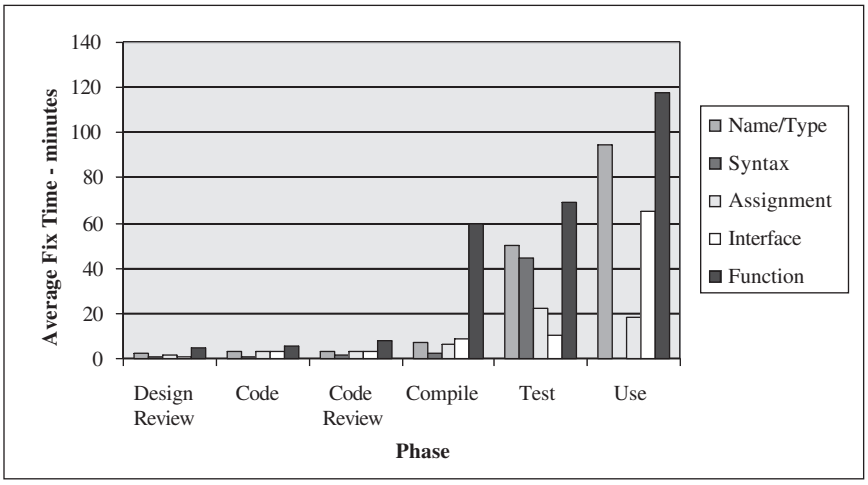


FIGURE 8.2 AVERAGE DEFECT FIX TIME (C++)

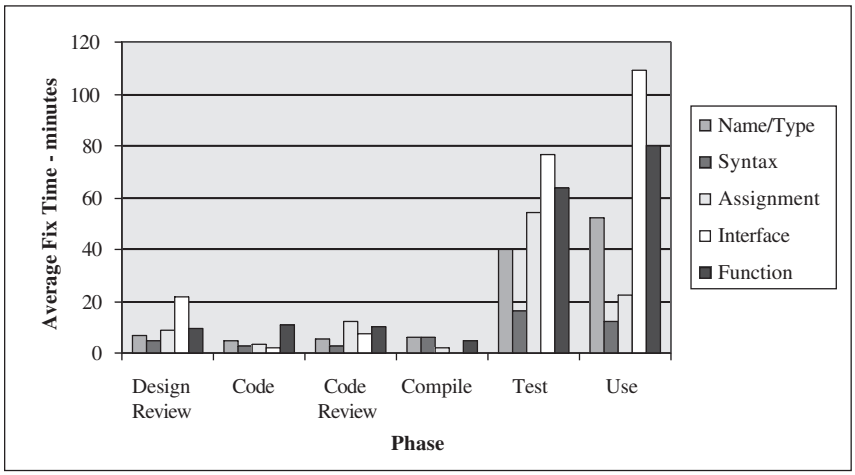


FIGURE 8.3 AVERAGE DEFECT FIX TIME (PASCAL)

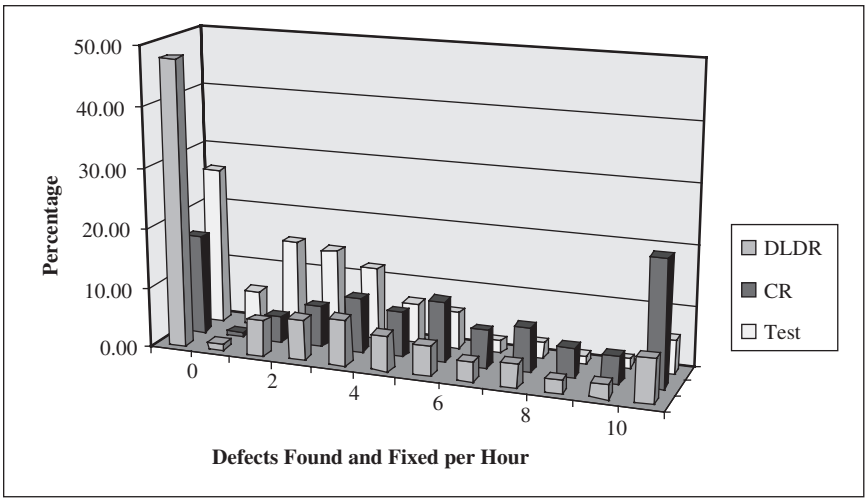


FIGURE 8.4 DEFECTS FOUND AND FIXED PER HOUR (3,240 PSP PROGRAMS)

Some people argue that the easy defects are found by inspections and the difficult ones are left for testing, but I have seen no data to support this. The PSP data show that, for every defect type and for every language measured, defect-repair costs are highest in testing and during customer use. Anyone who seeks to reduce development cost or time must focus on preventing or removing every possible defect before they start testing.

8.4 Defect Types

Defect tracking is important for all defect categories. Although developers will initially spend most of their time finding and fixing simple coding and design mistakes, their principal objective must be to deliver quality products. Therefore, they should be particularly concerned about the defects that they miss. Capers Jones, a well-known writer on software quality, reports that the distribution of delivered defects by type is as shown in Table 8.3. This table uses a conversion factor of 100 LOC per function point (Jones 2004).

The data in Table 8.3 show the importance of quality in every step of the development process. Although defective fixes account for only one in six of overall defect volume, they can be much higher for widely used products. With some IBM products, for example, fix errors accounted for over half of the customer-reported problems and an even higher percentage of service costs. The customer impact of defective fixes is high because many users install them shortly after receiving them. In other words, fix errors compound customer dissatisfaction because they affect users when they are already most unhappy with the product.

The data on error-prone modules also show why disciplined quality practices are so important. With IBM's OS/360 system, for example, only 4% of the modules accounted for over half of the customer-reported problems. For the IMS system,

TABLE 8.3 DELIVERED DEFECTS PER FUNCTION POINT AND KLOC

Defect Types/Origin	Defects/Function Point	Defects/KLOC
Requirements	0.23	2.3
Design	0.19	1.9
Coding	0.09	0.9
Document	0.12	1.2
Defective fixes	0.12	1.2
Total	0.75	7.5

only 31 of the 425 modules were responsible for 57% of the customer-reported defects (Jones 2004). When developers measure and manage the quality of their work, the inspection and test phases find many more of the requirements and design defects that are most important to users.

8.5 Personal Quality Practices

Most software professionals agree that it is a good idea to remove defects early, and they are even willing to try doing it. Beyond that, there is little agreement on how important this is. This is why, for example, developers will spend hours designing and coding a several-hundred-LOC program module and then spend only 10 or 15 minutes looking it over for any obvious problems. Although such superficial reviews may find something, the odds are against it. In fact, such cursory checks are usually a waste of time. To see how important it is to develop quality practices, and to begin to appreciate the enormous cost and schedule benefits of doing high-quality work, consider data on the numbers of defects in typical products and the costs of finding and fixing them.

Data for several thousand programs show that even experienced developers inject 100 or more defects per KLOC. Typically, about half of these defects are found during compiling and the rest must be found in testing. Based on these data, a 50,000-LOC product would start with about 5,000 defects, and it would enter testing with about 50 or more defects per KLOC, or about 2,500 defects. Again, about half of these defects would be found in unit testing at a rate of about two to three defects per hour, and the rest must be found in system-level testing at a cost of 10 to 20 or more hours each. Total testing times would then be about 13,000 to 25,000 hours. With roughly 2,000 working hours in a year, this would require 12 to 18 or more months of testing by a team of five developers.

With the PSP, you will use design reviews and code reviews to find defects before testing. Then, on a TSP team, you will also use inspections. When they do proper reviews and inspections, TSP teams typically find close to 99% of the defects before even starting system-level testing. Test times are then cut by ten times or more. Instead of spending months in testing, TSP teams need only a few weeks (Davis and Mullaney 2003).

If reviews and inspections are so effective, you might wonder why more organizations don't do them. There are two reasons. First, few organizations have the data to make sound quality plans. Their schedules are based largely on guesses, and their guesses are often wildly unrealistic. Then, when they treat these plans as accurate projections, there is no apparent reason to spend much time on reviews and inspections. However, during testing, the quality problems become clear, at

which point all the developers can do is test and fix until they get the product to work. The second reason why organizations typically do not do reviews and inspections is that without PSP data, the developers do not know how many defects they inject or what it costs to find and fix them. Therefore, neither they nor their organizations can appreciate the enormous benefits of finding and fixing essentially all of the defects before they start testing.

8.6 Quality Measures

To improve quality, you must measure quality. The key questions then are these: what do you measure and how do you use the resulting data? Before we can use the data, we must decide on the quality measures. The defect content of a finished product indicates the effectiveness of the process for removing defects. This, however, is an after-the-fact measure. To manage the quality of our work, we must measure the work, not just the work products. The available work measures are **defect-removal yield**, **cost of quality**, **review rates**, **phase-time ratios**, and the **process quality index (PQI)**.

The Yield Quality Measure

Figure 8.5 shows a filter view of the defect-removal process. By viewing defect removal as a sequence of filters that progressively remove defects, one can define useful defect-removal measures.

Yield measures the efficiency of each filtering phase at removing defects. The yield of a phase is the percentage of product defects that are removed in that phase. For example, if a product contained 20 defects at the start of unit testing and seven were found during testing, the unit test yield would be 100*7/20 = 35%. Yield, however, also considers the defects injected, as well as those removed in each phase. Thus, if you made a mistake while fixing the first 6 defects and injected a seventh defect, which you then found, you would have found 6 of the 20 defects present at phase entry and injected and fixed one more. The phase yield would then be based on finding 7 of 21 possible defects, for a 33% phase yield.

The yield of a phase, or **phase yield**, can be calculated for any phase. Thus, there is unit test yield, code review yield, compile yield, and so forth. There is also something called **process yield**, which refers to the percentage of defects that were removed before the first compile. Thus, if 100 defects were injected during requirements, design, and coding, and if 68 of them were found in all of the reviews and inspections up to but not including compile, the process yield would be 68%.

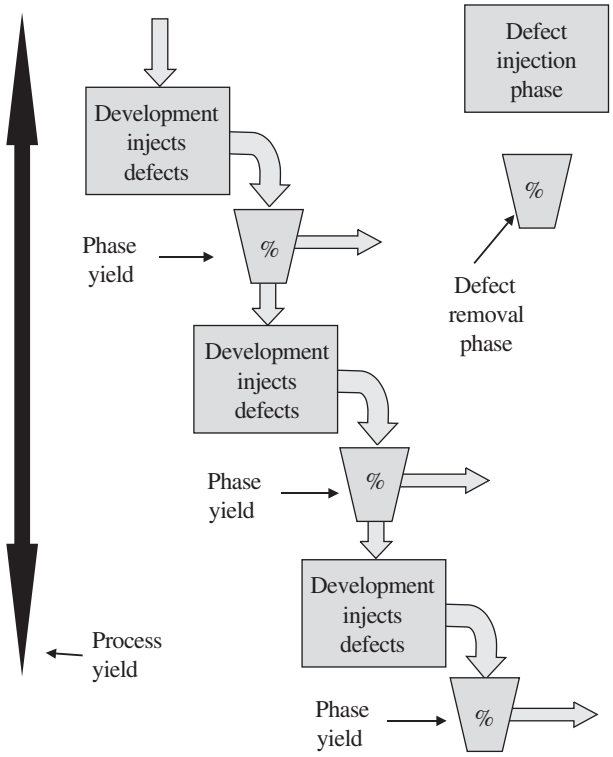


FIGURE 8.5 THE DEFECT-REMOVAL FILTERING PROCESS

Process yield can also be calculated for other phases, so if these same 100 defects had been injected in all of the phases up to system testing and 28 more defects were found in compiling, unit testing, and integration testing, then 68 + 28 = 96 defects would have been found before starting system testing. The process yield before system testing would then be 96%. When used without a phase name, process yield refers to the yield before compiling. If your development environment does not have a compile phase, then process yield will be measured up to but not including unit testing. Otherwise, when the phase name is included, phase yield can be calculated for any phase. The process yield, or yield-before-compile for a typical PSP class, is shown in Figure 8.6. Although most developers start with low process yields, their yields increase sharply when they start using design and code reviews.

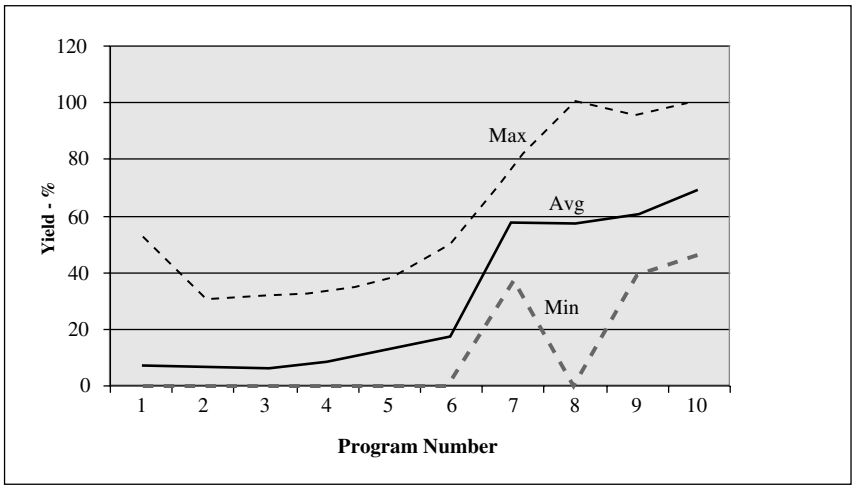


FIGURE 8.6 YIELD FOR 12 DEVELOPERS ON 10 PSP EXERCISES

The Cost of Quality (COQ)

The cost-of-quality measure should more properly be called the cost of poor quality. Juran describes **cost of quality** as a way to "quantify the size of the quality problem in language that will have impact on upper management" (Juran and Gryna 1988). COQ thus defines quality in cost terms that can easily be related to other aspects of a business. The cost of quality has three principal components: failure costs, appraisal costs, and prevention costs (Crosby 1983; Mandeville 1990; Modarress and Ansari 1987). Their definitions are as follows:

1. **Failure costs:** The costs of diagnosing a failure, making necessary repairs, and getting back into operation
2. **Appraisal costs:** The costs of evaluating the product to determine its quality level
3. **Prevention costs:** The costs of identifying defect causes and devising actions to prevent them in the future

To strictly follow these COQ definitions, 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 counted as failure costs. For the PSP, however, we use somewhat simpler definitions:

- **Failure costs for the PSP:** The total cost of compiling and testing. Because defect-free compile and test times are typically small compared to total compile and test times, they are included in failure costs.
- **Appraisal costs for the PSP:** The total times spent in design and code reviews and inspections. Inspections and design and code reviews are covered in Chapter 9. Because defect-repair costs are typically a small part of review and inspection costs, the PSP includes them in appraisal costs.

If the inspection and review fix times or the defect-free compile and test times were large enough to consider, however, you could measure them and use more precise cost-of-quality definitions. For personal process-management purposes, however, the simpler definitions are easier to track and equally useful.

The costs of prototype development, causal analysis meetings, and process improvement action meetings should be classified as prevention costs. The times spent in improving design methods, defining coding standards, and initiating personal defect tracking are all effective at reducing defect-injection rates, and are defect-prevention activities. The PSP does not provide measures for defect prevention, however, as this work typically involves cross-team and cross-project activities. To be most effective, defect-prevention work should span multiple projects and be incorporated into organization-wide processes and standards. The PSP 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)
- **Appraisal to Failure Ratio (A/FR):** (Appraisal COQ)/(Failure COQ)

The A/FR measure is useful for tracking personal process improvement. Figure 8.7 shows the A/FR results for 12 developers on 10 PSP exercises. Figure 8.8 shows the relationship of A/FR to test defects for these same developers. The number of test defects is typically much lower for higher values of A/FR. Although a high A/FR implies lower defects, too high a value could mean that you are spending excessive time in reviews. The PSP guideline is an A/FR of about 2.0.

Review Rate Measures

Although yield and COQ measures are useful, they measure what you did, not what you are doing. To do quality work, we need measures to guide what we do while we are doing it. In design or code reviews, the principal factors controlling yield are the time and the care a developer takes in doing the review. The review

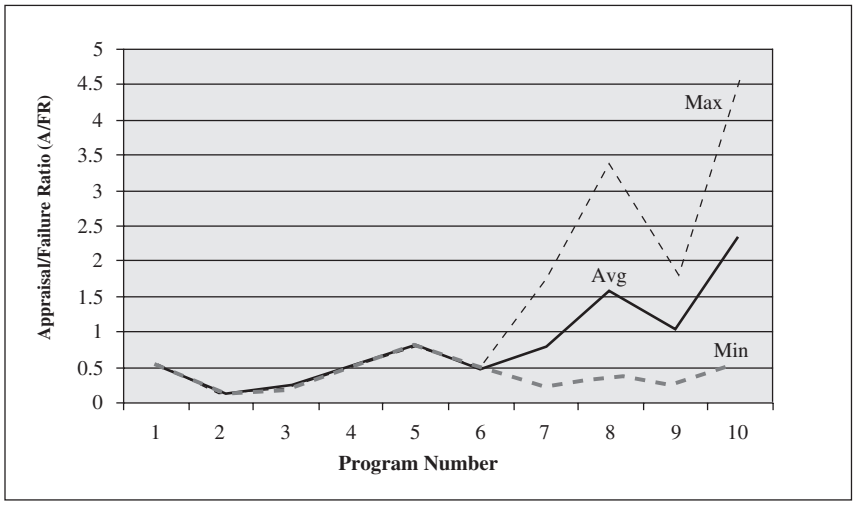


FIGURE 8.7 A/FR (12 DEVELOPERS)

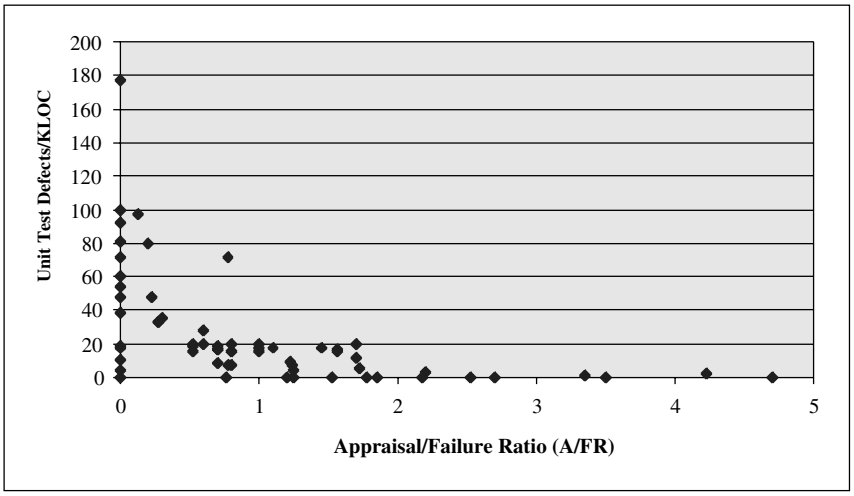


FIGURE 8.8 A/FR VERSUS UNIT TEST DEFECTS (12 DEVELOPERS)

rate and phase ratio measures provide a way to track and control review times. The review rate measure is principally used for code reviews and inspections, and measures the LOC or pages reviewed per hour. If, for example, you spent 20 minutes reviewing a 100-LOC program, your review rate would be 300 LOC per hour. There is no firm rate above which review yields are bad. However, the PSP data show that high yields are associated with low review rates, and that low yields are typical with high review rates. Figure 8.9 shows yield data for 810 developers.

The front set of bars is a histogram of yield for low review rates, and the back bars represent high review rates. The trends can be seen most clearly by looking at the bars for zero-yield reviews at the left and 100%-yield reviews at the right. The highest review rate had the highest percentage of zero-yield reviews. A zero-yield review is a high-tech way to waste your time. The PSP rule of thumb is that 200 LOC per hour is about the upper limit for effective review rates and that about 100 LOC per hour is recommended as the target review rate. The best guideline is to track your own review performance and find the highest rate at which you can review and still consistently get review yields of 70% or better.

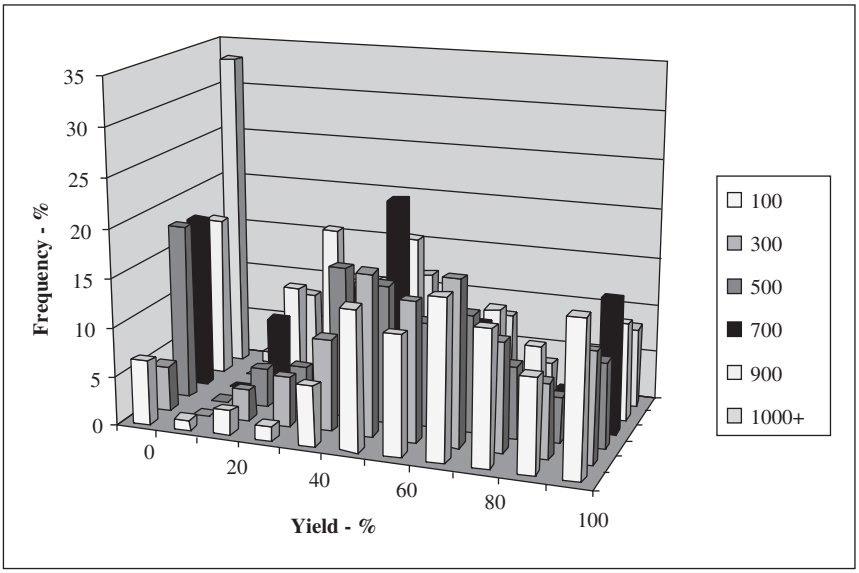


FIGURE 8.9 YIELD VERSUS REVIEW RATE (810 DEVELOPERS)

Phase Ratio Quality Measures

Another set of useful quality measures is the ratio of the time spent in two process phases. For example, the ratio of design review time to design time. To measure process quality, the PSP uses the ratios of design to coding time, design review to design time, and code review to coding time. One would expect that increases in design time would tend to improve product quality while correspondingly reducing coding time. The data in Figure 8.10 for 810 developers show that this is generally the case. The front bars represent developers who spent half as much time designing as coding; the back represents developers who spent twice as much time designing as coding.

Although there is no clear cutoff point below which quality is poor, a 1-to-1 ratio of design to coding time appears to be a reasonable lower limit, with a ratio of 1.5 being optimum. Again, this optimum point varies widely by individual and program type, but a useful rule of thumb is that design time should at least equal coding time. If it doesn't, you are probably doing a significant amount of design work while coding. As shown in Table 8.4 (p. 140), because developers typically inject more than twice as many defects per hour in coding, designing while coding is not a sound quality practice.

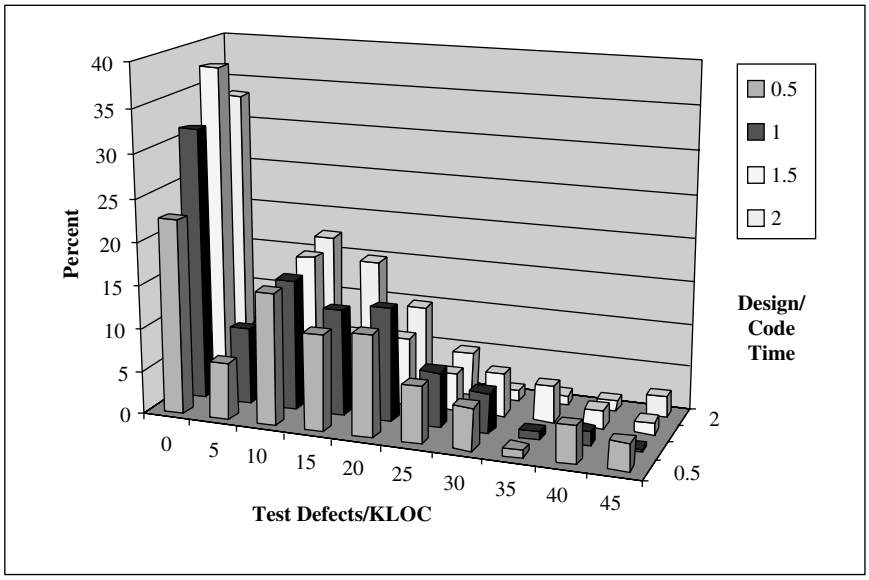


FIGURE 8.10 DEFECTS/KLOC VERSUS DESIGN/CODE TIME (810 DEVELOPERS)

TABLE 8.4 DEFECT-INJECTION AND -REMOVAL RATES (3,240 PSP PROGRAMS)

Phase	Hours	Defects Injected	Defects Removed	Defects/Hour
Design	4,623.6	9,302		2.011
DLDR	1,452.7		4,824	3.321
Code	4,159.6	19,296		4.639
Code Review	1,780.4		10,758	6.043

Another useful ratio is for review time divided by development time. In the PSP, the general guideline is that you should spend at least half as much time reviewing a product as you spent producing it. This means that for every hour of coding, for example, you should spend at least half an hour doing a code review. The same ratio holds for design and design review time, requirements and requirements review time, and so forth.

One way to determine the reasonableness of these review ratios is by comparing defect-injection and -removal rates. Table 8.4 shows the defect-injection rates for detailed design and coding for the same 810 developers on the 3,240 programs they wrote with PSP2 and PSP2.1. The developers injected 9,302 defects in 4,624 hours of design and injected 19,296 defects in 4,160 hours of coding. These rates are 2.01 defects injected per hour in detailed design and 4.64 defects injected per hour during coding. The table also shows the defect-removal rates for these same developers in design reviews and code reviews. Here, the average removal rates are 3.32 defects per hour in design review and 6.04 defects per hour in code review. The ratios of these rates indicate that these developers must spend at least 76.8% of their coding time in code reviews to expect to find the defects they inject. Similarly, they must spend at least 60.5% of their design time in design reviews to find their design defects.

The PSP guideline of 50% is sufficiently close to these values to provide a simple and easy-to-remember target. However, you should view it as a minimum, with the optimum closer to 70%. As you gather data on your own performance, select the guideline that best fits your personal data.

The Process Quality Index (PQI)

Suppose that you followed all of these guidelines, that you spent as much time designing as coding, and that you met the 50% guidelines for design and code review times. Would this guarantee high quality? Unfortunately, no. Success depends on how you spend your time. Did you follow sound design practices and perform high-yield reviews?

These questions imply the need for at least two other measures. The first is to evaluate the quality of your code reviews. If you followed the code-review process (see Chapter 9) and used an up-to-date code-review checklist, you would probably have a high-yield review. The best indicator of this would be finding very few defects while compiling. In the PSP, 10 or fewer defects per KLOC during compiling is considered low. Without a pretty good code review, you are unlikely to get this low a number. Unfortunately, if your environment doesn't use a compile phase, this measure is not available. In these cases, you can use the data on the coding-type defects found in unit testing and, on a TSP team, the coding-type defects found during inspections and unit testing. For these purposes, defect types 10 through 50 can be considered coding-type defects.

The design review measure is somewhat more complex. Although the number of defects found in unit testing could be a useful measure of design quality, it also includes code quality. If, however, your compile defects/KLOC were below 10, then code quality is probably pretty good and the unit test defects would measure design quality. Thus, without data on compile defects, you cannot interpret the unit test defect measure. The suggested PSP measure specifies that if the compile defects/KLOC are less than 10, unit test defects/KLOC of 5 or less indicates a quality design. The process quality index (PQI) quality criteria are as follows:

- Design/Code Time = design time/coding time, with a range from 0.0 to 1.0
- Design Review Time = 2*design review time/design time, with a range from 0.0 to 1.0
- Code Review Time = 2*code review time/coding time, with a range from 0.0 to 1.0
- Compile Defects/KLOC = 20/(10 + compile defects/KLOC), with a range from 0.0 to 1.0
- Unit Test Defects/KLOC = 10/(5 + unit test defects/KLOC), with a range from 0.0 to 1.0

The process quality index is obtained by multiplying these five values together. Figure 8.11 shows the PQI profiles for six programs. The two programs with the poorest-quality profiles (numbers 5 and 6) were the only ones with defects after unit testing. The data in Figure 8.12 show that programs with PQI values above 0.4 historically have had no defects found after unit testing. The PQI goal, however, should be 1.0.

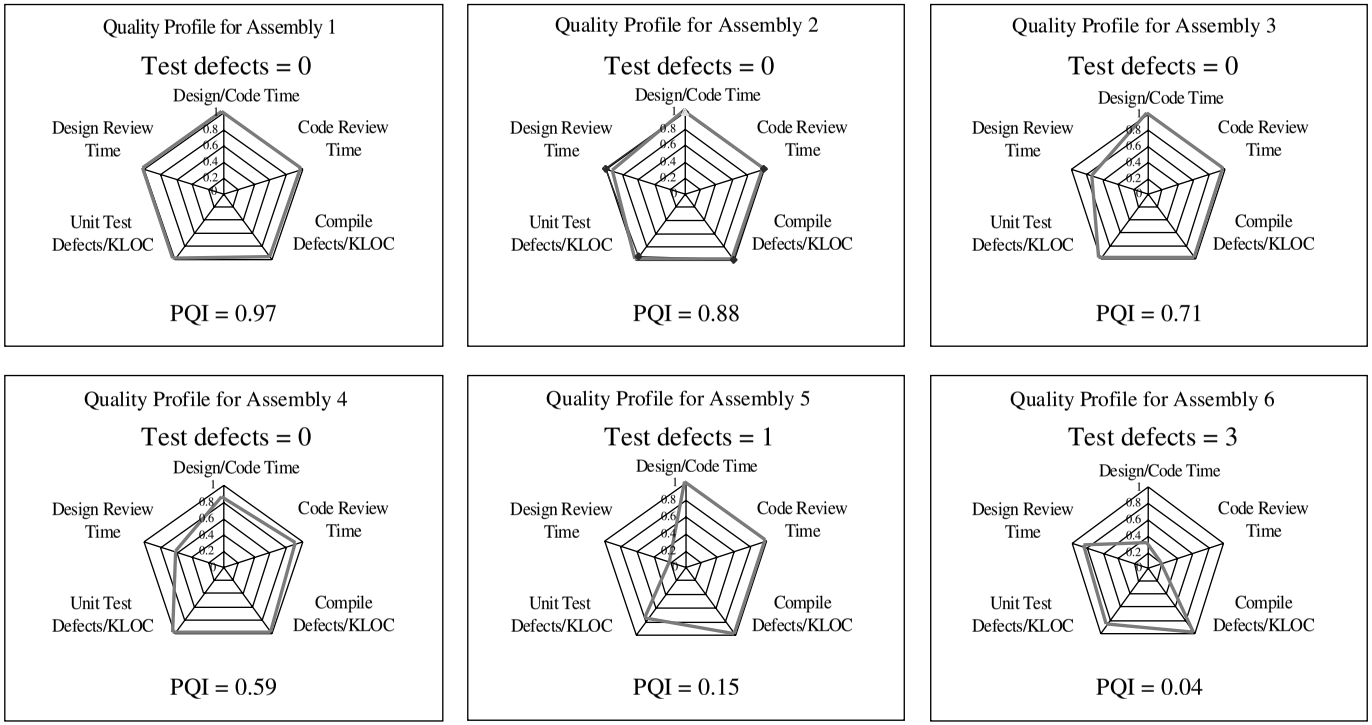


FIGURE 8.11 PROCESS QUALITY INDEX (6 PROGRAMS)

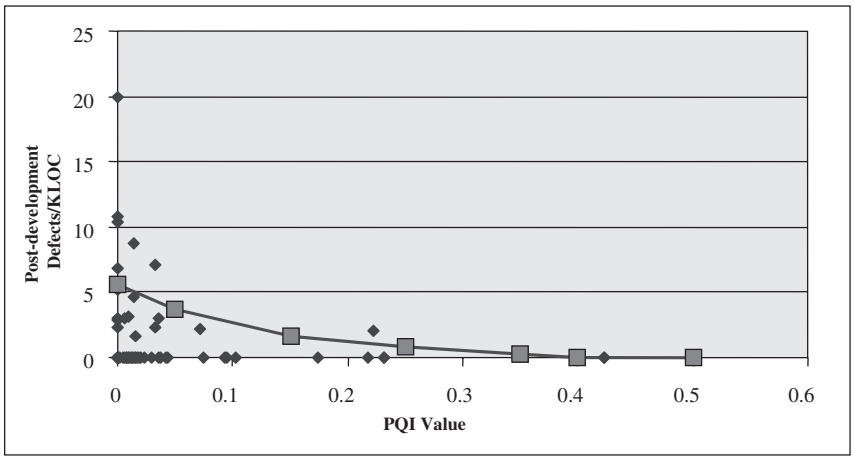


FIGURE 8.12 PQI VALUES VERSUS POST-DEVELOPMENT DEFECTS

8.7 Quality Management

It takes extraordinary skill and discipline just to get a reasonably sized program to run. Although the program may have remaining defects, the fact that it runs at all is an achievement; and though we may celebrate this feat, society now needs higher-quality software than individuals normally produce. We therefore need methods for producing high-quality products even though we personally inject defects. This requires measures to guide our improvement actions.

One can view the software process as the combination of two competing processes: defect injection and defect removal. The defect content of the finished product comes from the difference between these two processes. Thus, relatively small changes in process yield can cause large changes in the defect content of the finished product.

For example, in developing a million-LOC program, the developers would likely inject about 100 defects per KLOC, or 100,000 defects. If the process yield was 75%, 25,000 defects would be left for compiling and testing. If, again, as the PSP data show, about half of these defects were found during compiling, 12,500 defects would be left to find in testing. With four levels of testing, each with a 50% yield, 781 defects would still remain in the delivered product, for a defect level of 0.8 per KLOC. This is a rather good level of quality by current industrial standards. Suppose, however, that the process yield fell to 50%. Then 50,000 defects

would remain for compiling and testing. If the yields of the other process steps remained the same, the final number of defects in the delivered product would double to 1,562, or 1.6 defects per KLOC. Thus, a 33% reduction in yield would double the number of delivered defects.

Although this is a significant increase in the number of defects shipped to the customer, the customer impact is modest compared to the impact on testing. If, as the data show, the average cost to remove defects in testing is 10 or more hours per defect, the 12,500 defects originally left for testing would take 125,000 or more programmer hours, or about 60 programmer years. With the reduced inspection yield of 50%, this number would double to 25,000 defects and 120 programmer years. Thus, a 33% reduction in review and inspection yields would cause 50 to 60 developers to spend another year testing and fixing the product. Even with this extra year, the final product would still have twice as many defects as with the original 75% yield.

From a yield-management point of view, the final number of defects is the residual of the yields of all of the process phases. If any single phase for any product component has low yield, some percentage of the defects it misses will impact every subsequent phase and degrade the final product. This degradation will also impact the costs of maintenance, of support, and of using the product, as well as the costs of all subsequent product enhancements. Poor-quality work has a lasting effect, so it is important that every developer manage the quality of his or her personal work. Not only will quality work save you time and effort, it will produce savings throughout the product life cycle.

8.8 Personal Quality Management

To effectively manage software quality, you must focus on both the defect-removal and the defect-injection processes. For example, some days you might inject a lot of defects because you are sick, have personal problems, or did not get enough sleep. Regardless of your training, skill, or motivation, there is a distinct probability that any action you take will produce an error. Even when I am extremely careful, close to 5% of all my changes contain mistakes that later require correction. This is true whether I am writing programs, doing financial calculations, or editing a manuscript. I have thus learned to expect such errors and to check my corrections and keep checking them until I find no more errors.

For software, the change process is the most error-prone activity (Levendel 1990). The percentage of fixes that generate defects is rarely less than 20% and it is generally much higher. When people recognize this problem and work to address it, they can bring the percentage of fix errors down to 5% or less (Bencher 1994).

Some of the most error-prone programming actions involve interpreting requirements and making design decisions. Errors are also likely during logic design

and coding. Some percentage of fix decisions will be in error, as will a fraction of all keystrokes. Even expert typists make mistakes, though they make no design decisions. The quality strategy must therefore recognize that every action has a probability of injecting a defect. The challenge is to identify the defect sources and to prevent them, or to devise actions to consistently detect and remove all the defects that slip through.

The PSP's recommended personal quality-management strategy is to use your plans and historical data to guide your work. That is, start by striving to meet the PQI guidelines. Focus first on producing a thorough and complete design and then document that design with the four PSP design templates that are covered in Chapters 11 and 12. Then, as you review the design, spend enough time to find the likely defects. If the design work took four hours, plan to spend at least two, and preferably three, hours doing the review. To do this productively, plan the review steps based on the guidelines in the PSP Design Review Script described in Chapter 9. If you finish all of the steps in less than the planned time, check to see if you skipped anything or made any mistakes. Then, in the code review, follow the PSP Code Review Script described in Chapter 9. Make sure that you take the time that your data say you must take to do a quality job.

Although it is most desirable to find and fix all defects before testing, this is rarely possible. Reviews and inspections are performed by people, and people make mistakes. Because we cannot always expect to reach 100% yield in code reviews or inspections, testing must be part of every software quality strategy. But reviews, inspections, compiling, and testing are each most effective at finding different kinds of defects. For example, in design reviews, people will quickly see problems that would be extraordinarily hard to find in testing. Similarly, some system problems that are very hard to visualize in a review or inspection show up right away in testing.

In practice, this means that when you do careful personal reviews, you will find most of the defects and find them rapidly—but you will miss some. Then, when your team does thorough design and code inspections, the inspectors will find many of the defects that you missed. Some will be mistakes and others could be errors that you could not see because, every time you looked, you saw what you meant to do, rather than what you actually did.

The compiler will find other types of defects, as will every phase of testing. Surprisingly, however, when developers and their teams use reasonable care throughout the process, their finished products have essentially no defects. Of course, there is no way to know for certain that these products are defect-free, but many TSP teams have delivered products for which users have not reported defects. In short, quality work pays in reduced costs, shorter schedules, and higher-quality products.

What is right for you today will not likely be best forever. Not only will you develop new skills, but the technology will change, you will have a different working environment, and you will face different problems. You must thus continually

track and assess the quality of your work. When improvement progress slows, set new goals and start again. Be alert for new techniques and methods, and use your data to determine what works best for you.

8.9 Managing Product Quality

The software quality problem is becoming more challenging every year. Initially, the issue was to get programs to run. That involved running a few tests until the program did what it was supposed to do. As systems have become more complex, the testing workload has grown until it now consumes about half of the development schedule. The PSP quality strategy addresses this issue by removing as many defects as possible before testing. This strategy has improved both product quality and development productivity. A study of TSP team results found that quality improvements of one to two orders of magnitude were common. Figure 8.13 shows the defect levels typical of shipped products at organizations at various levels of CMM maturity, together with the average defect level of the products shipped by a couple of dozen TSP teams (Davis and Mullaney 2003). Because

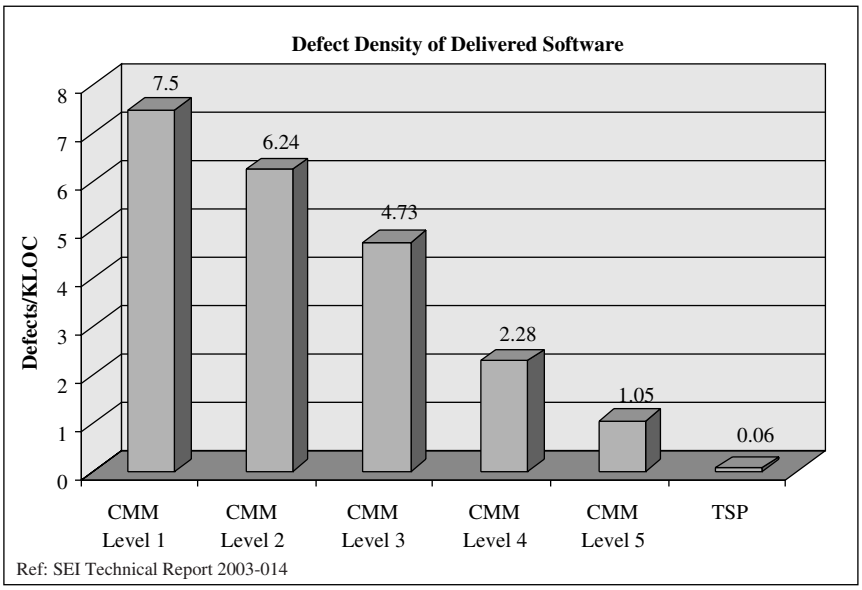


FIGURE 8.13 CMM AND TSP DEFECT LEVELS

most of the TSP teams were in organizations at CMM levels 1 and 2, the 0.06 average defect/KLOC was an improvement of about 100 times.

The challenge we face is that these quality levels are not adequate. The increasing importance of safety and security and the growing use of software to control safety and security in critical systems means that one defect per KLOC is not acceptable. In fact, even 100 defects per million lines of code (MLOC) is potentially dangerous. With today's multimillion-LOC systems, we must strive for 10 defects per MLOC or less. The question is, how can we possibly do this?

As noted earlier, this would require that developers miss only one single defect in 3,000 pages of listings, so we must devise the highest-quality processes we can create; and then we must measure and improve these processes so that the likelihood of having one defect in a one-KLOC module is less that 1%. That is what the PQI measure is designed to accomplish. The strategy is for you to follow a high-quality personal process with every product element. Then, your TSP team should follow a measured and managed team process to inspect and test that product element before including it in the system. When the data for any module look questionable, the team pulls it and reexamines it to determine whether it should be reinspected, reworked, or even completely replaced. What is exciting is how well this strategy works. Forty percent of the TSP teams have reported delivering defect-free products. And these teams had an average productivity improvement of 78% (Davis and Mullaney 2003). Quality isn't free—it actually pays.

8.10 PSP Improvement Practices

In software development, we rarely have the opportunity to develop brand-new products. Most of our work involves enhancing and fixing existing products, and many of these products have unknown and questionable quality histories. A question that I am often asked is, "How can I use the PSP on this kind of work?" In considering this question, the first point to remember is that the PSP is based on six principles:

1. To have predictable schedules, you must plan and track your work.
2. To make accurate and trackable plans, you must make detailed plans.
3. To make accurate detailed plans, you must base the plans on historical data.
4. To get the data needed to make accurate plans, you must use a defined and measured personal process.
5. To do high-quality work, you must measure and manage the quality of your development process.
6. Because poor-quality work is not predictable, quality is a prerequisite to predictability.

Although the PSP course focuses on how to apply these principles when developing new or modified module-size programs, the principles are equally applicable to modifying and enhancing large existing systems. In fact, these same principles also apply to developing requirements, designing hardware devices, and writing books.

The PSP principles can be used and are very effective for maintaining and enhancing existing products, but the quality benefits are more variable. The governing factors are the size of the existing product, the nature of the enhancement work, the historical data available, the quality of the existing product, the goals for the work, and the skills and resources available. There are many ways to design processes for maintenance and enhancement work but the example processes in Chapter 13 suggest ways to define such a process to best fit your needs.

8.11 Defect Prevention

Although detecting and fixing defects is critically important, it is an inherently defensive strategy. For the quality levels required today, we must identify the defect causes and devise steps to prevent them. By reviewing the data on the defects you and your team find during development and testing, you can establish a personalized defect-prevention effort. Possible strategies are to focus on defect types in any of the following categories:

- Those found in final program testing or use
- Those that occur most frequently
- Those that are most difficult or expensive to find and fix
- Those for which you can quickly identify preventive actions
- Those that most annoy you

The TSP process suggests a procedure for doing this through analyzing every defect found after unit testing. The key is to review every test and user-reported defect, gather the essential data, and establish an explicit prevention strategy. Then, start with a narrow defect category. Once you see what works best, you will have a better idea of how to proceed. The focus in reviews is on finding and fixing defects. Defect prevention, however, requires a more detailed study of the defects. You need to determine what caused them before you can devise ways to prevent them. Then you must work to consistently prevent these defects in the future. A suggested approach is as follows:

1. Select a defect type or class for analysis.
2. Summarize the causes of the defect. In team reviews, it is often helpful to use

Ishikawa or wishbone diagrams (Schulmeyer and McManus 1999). I have not found them necessary for the PSP but you may want to try them. An example is shown in Figure 8.14.

3. Look for trends or patterns in the data that suggest larger or more pervasive problems. An example might be a class of problem that gives you trouble right after an extended period of nonprogramming work.
4. Now that you understand the errors that caused the defects, determine why you made them. Some potential causes are poor or inadequate communication, education gaps, transcription errors, and incomplete or imprecise processes.
5. Devise ways to prevent these problems in the future.
6. Analyze actions that have worked in the past and ensure that you retain them and build on them.
7. Test your defect-prevention ideas on at least one PSP-size program and incorporate those that work in a process update.
8. Recycle through steps 1 through 7 until you run out of time or have no more useful ideas.

Although this approach is effective at a team or project level, it can be difficult to implement by yourself (Gale et al. 1990; Mays et al. 1990). One way to proceed when you run out of good ideas is to hold occasional defect-prevention brain-

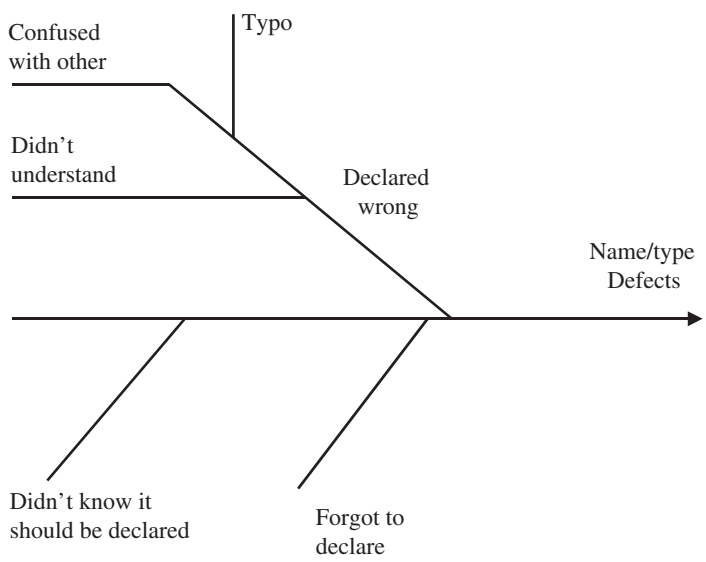


FIGURE 8.14 ISHIKAWA DIAGRAM

storming sessions with some team members. As you get more ideas, consider forming a defect-prevention group.

Try to hold these sessions on a set schedule, with the frequency controlled in part by how much software development you and your teammates do. It probably should not be more often than monthly and no less often than quarterly. If you do action planning too often, you will not have enough time to gauge the effectiveness of your previous actions. Conversely, if you wait too long, your progress will be slower and you are less likely to remember the defect causes. Therefore, do your first defect-prevention studies as soon as you have enough data to analyze. Data on 15 to 30 system-test or user-generated defects should provide enough data for useful analysis. Waiting much longer risks getting overburdened with data and unnecessarily fixing defects you could have prevented with a more timely review.

After implementing the resulting actions, be guided by your ideas. When you think another session would be productive, even if it is the next day, do it. When you run out of ideas, establish a schedule for the next review. Initially, set it a month away. Then, if the monthly reviews become less productive, space them further apart. Don't waste time holding a defect-prevention meeting if you don't have any new ideas. However, do not stop doing them entirely. Again, if you can get one or more teammates to help, you will be more productive. To start a defect-prevention effort, do the following.

1. Use your PSP data to establish a priority for the defect types to initially address. First pick a narrow class of problems and define explicit actions to prevent them.
2. Change your process to accomplish these actions. These changes may include changing forms, adding checklist items, or modifying standards. When practical, devise automated aids to help with transcription and oversight problems. Other automated defect-prevention actions will likely involve too much work to implement by yourself.
3. Conduct a prototype walk-through of the preventive measures to ensure that they are convenient, sustainable, and effective. Then make the indicated changes and insert the new methods into your standard process.

8.12 Summary

To improve the quality of your software, focus on the process required to consistently produce quality products. Seek the most effective methods for finding defects and the most effective ways to prevent them. The costs of finding and fixing defects escalate rapidly the longer the defects remain in the product, and the most

cost-effective strategy is to ensure that all program elements are of the highest quality when they are first produced.

Although defects are only one facet of software quality, that is the quality focus of this book. Defects are rarely the users' top priority, but they are an essential focus of the PSP. This is because defects are most economically and effectively handled at the individual level. If the elemental programs in a system have many defects, the entire development process will be overwhelmed by the time-consuming and expensive process of finding and fixing defects during testing.

Software quality is an economic issue. You can always run another test or do another inspection. However, few organizations have the data to make sound quality plans. The PSP provides the data needed to calculate measures for yield, cost of quality, rates and ratios, and the process quality index (PQI). Process yield refers to the percentage of total defects removed before compiling and testing, and the cost-of-quality measure quantifies the quality problem in economic terms. The principal cost-of-quality elements are failure costs, appraisal costs, and prevention costs. These data are used to measure and evaluate the quality of your development process and to identify ways to improve it.

The software process can be viewed as the combination of two competing processes: defect injection and defect removal. The defect content of the finished product is then governed by the difference between the output of these two processes. Because the result is the difference of two large numbers, relatively small changes in either process can make a large difference in the final result. To effectively manage software quality, you must focus on both the removal and the injection processes. Although detecting and fixing defects is critically important, it is an inherently defensive strategy. To make significant quality improvements, identify the causes of the defects and then take steps to eliminate the causes, thus preventing the resulting defects.

References

Ackerman, A. F., L. S. Buchwald, and F. H. Lewski. "Software Inspections: An Effective Verification Process." *IEEE Software* (May 1989): 31–36.

Bencher, D. L. "Technical Forum." *IBM Systems Journal* 33, no. 1 (1994).

Boehm, B. W. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

Bush, M. "Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory." Twelfth International Conference on Software Engineering, Nice, France (March 26–30, 1990): 196–199.

Crosby, P. B. *Quality Is Free: The Art of Making Quality Certain.* New York: New American Library, 1979.

———. "Don't Be Defensive about the Cost of Quality." *Quality Progress* (April 1983).

Davis, N., and J. Mullaney. "Team Software Process (TSP) in Practice." SEI Technical Report CMU/SEI-2003-TR-014 (September 2003).

Gale, J. L., J. R. Tirso, and C. A. Burchfield. "Implementing the Defect Prevention Process in the MVS Interactive Programming Organization." *IBM Systems Journal* 29, no. 1 (1990).

Jones, C. Personal e-mail, 2004.

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

Levendel, Y. "Reliability Analysis of Large Software Systems: Defect Data Modeling." *IEEE Transactions on Software Engineering* 16, no. 2 (February 1990): 141–152.

Mandeville, W. A. "Software Costs of Quality." *IEEE Journal on Selected Areas in Communications* 8, no. 2 (February 1990).

Mays, R. G., C. L. Jones, G. J. Holloway, and D. P. Studinski. "Experiences with Defect Prevention." *IBM Systems Journal* 29, no. 1 (1990).

Modarress, B., and A. Ansari. "Two New Dimensions in the Cost of Quality." *International Journal of Quality and Reliability Management* 4, no. 4 (1987): 9–20.

O'Neill, D. Personal letter, 1994.

Ragland, B. "Inspections are Needed Now More Than Ever." *Journal of Defense Software Engineering* 38. Software Technology Support Center, DoD (November 1992).

Remus, H., and S. Ziles. "Prediction and Management of Program Quality." *Proceedings of the Fourth International Conference on Software Engineering, Munich, Germany* (1979): 341–350.

Russell, G. W. "Experience with Inspections in Ultralarge-Scale Developments." *IEEE Software* (January 1991): 25–31.

Schulmeyer, G. G., and J. I. McManus. *Handbook of Software Quality Assurance.* New York: Van Nostrand, 1999.

Shooman, M. L., and M. I. Bolsky. "Types, Distribution, and Test and Correction Times for Programming Errors." *Proceedings of the 1975 Conference on Reliable Software, IEEE, New York,* catalog no. 75, CHO 940-7CSR: 347.

vanGenuchten, M. Personal conversation, 1994.

Weller, E. F. "Lessons Learned from Two Years of Inspection Data." *IEEE Software* (September 1993): 38–45.