Software Engineering Institute | Carnegie Mellon

The Software Quality Profile

Watts S. Humphrey

Abstract

The software community has been slow to use data to measure software quality. This paper discusses the reasons for this problem and describes a way to use process measurements to assess product quality. The basic process measures are time, size, and defects. When these data are gathered for every engineer and for every step of the development process, a host of quality measures can be derived to evaluate software quality. Extensive data from the Personal Software Process^{SM} (PSP^{SM}) are used to derive the profiles of software processes that generally produce high quality software products. By examining these profiles, one can judge the likelihood that a program will have defects found in its subsequent testing or use. Examples are given of defect profiles, together with guidelines for their use.

Introduction

Without numbers, quality programs are just talk.

There are four reasons why the software community has been slow to use numbers for software quality.

1. There is no generally recognized definition for quality measures.
2. Even when we know what numbers to use, the data are not easy to gather.
3. Even with data, it is not obvious how to interpret and use the numbers.
4. People are reluctant to measure the quality of their personal work.

This paper addresses these questions and gives examples of software quality data that can readily be gathered and used by properly trained engineers. While we focus on measuring and controlling the defect content of programs, other quality measures are important to customers. We must, however, measure and control defect content before other quality aspects can be effectively addressed. Some of the material in this paper is taken from two textbooks I have written to introduce process methods in undergraduate and graduate software courses [Humphrey 95, Humphrey 97]. The rest comes from some as yet unpublished work with development teams.

The Software Quality Problem

Software quality is becoming increasingly important. Software is now used in many demanding applications and software defects have caused serious damage and even physical harm. While defects in financial or word processing programs are annoying and possibly costly, nobody is killed or injured. When software-intensive systems fly airplanes, drive automobiles, control air traffic, run factories, or operate power plants, defects can be dangerous. People have been killed by defective software [Leveson 95].

While there have not been many fatalities so far, the numbers will almost certainly increase. In spite of all its problems, software is ideally suited for critical applications: it does not wear out or deteriorate. Computerized control systems are so versatile, economical, and reliable that they are now the common choice for almost all systems. Software engineers must thus consider that their work could impact the health, safety, and welfare of many people.

The Risks of Poor Quality

Any defect in a small part of a large program could potentially cause serious problems. While it may seem unlikely that a simple error in a remote part of a large system could be potentially disastrous, these are the most frequent sources of trouble. The problem is that systems are becoming faster, more complex, and automatic. Catastrophic failures thus are increasingly likely and potentially more damaging [Perrow 84].

In the design of large systems, the difficult design issues are often carefully studied, reviewed, and tested. As a result, the most common causes of software problems are simple oversights and goofs.

These are typically simple mistakes that were made by individual software engineers. While most of these simple mistakes will get caught in compiling and testing, engineers inject so many defects that large numbers still escape the entire testing process and are left to be found during product use.

The problem is that software engineers often confuse simple with easy. They feel that their frequent simple mistakes will be simple to find. They are often surprised to learn that such trivial errors as omitting a punctuation mark, misnaming a parameter, incorrectly setting a condition, or misterminating a loop could escape testing and cause serious problems in actual use. These, however, are the kinds of things that cause a large proportion of the problems software suppliers spend millions of dollars finding and fixing. While most of these trivial mistakes will have trivial consequences, a few can cause unpredictable and possibly damaging problems.

The quality of large programs depends on the quality of the smaller programs of which they are built. Thus, to produce high quality large programs, every software engineer who develops one or more of the system's parts must do high-quality work. This means that all the engineers must be personally committed to quality. When they are so committed, they will track and manage their defects with such care that few if any defects will later be found in integration, system testing, or by the customers. The SEI has developed the Personal Software Process (PSP) and the Team Software Process^{SM} (TSP^{SM}) to help engineers work this way.

Measuring Software Quality

Software quality impacts development costs, delivery schedules, and user satisfaction. Because software quality is so important, we need to first discuss what we mean by the word ***quality***. The quality of a software product must be defined in terms that are meaningful to the product's users. What is most important to them and what do they need?

Defects and Quality

A software engineer's job is to deliver quality products for their planned costs, and on their committed schedules. Software products must also meet the user's functional needs and reliably and consistently do the user's job. While the software functions are most important to the program's users, these functions are not usable unless the software runs. To get the software to run, however, engineers must remove almost all its defects. Thus, while there are many aspects to software quality, the first quality concern must necessarily be with its defects.

The reason defects are so important is that people make a lot of mistakes. In fact, even experienced programmers typically make a mistake for every seven to ten lines of code they develop [Humphrey 96]. While they generally find and correct most of these defects when they compile and test their programs, they often still have a lot of defects in the finished product.

What Are Defects?

Some people mistakenly refer to software defects as bugs. When programs are widely used and are applied in ways that their designers did not anticipate, seemingly trivial mistakes can have unforeseeable consequences. As widely used software systems are enhanced to meet new needs, latent problems can be exposed and a trivial-seeming defect can truly become dangerous. While the vast majority of trivial defects have trivial consequences, a small percentage of seemingly silly mistakes can cause serious problems. Since there is no way to know which of these simple mistakes will have serious consequences, we must treat them all as potentially serious defects, not as trivial-seeming "bugs."

The term defect refers to something that is wrong with a program. It could be a misspelling, a punctuation mistake, or an incorrect program statement. Defects can be in programs, in designs, or even in the requirements, specifications, or other documentation. Defects can be redundant or extra statements, incorrect statements, or omitted program sections. A defect, in fact, is anything that detracts from the program's ability to completely and effectively meet the user's needs. A defect is thus an objective thing. It is something you can identify, describe, and count.

Simple coding mistakes can produce very destructive or hard-to-find defects. Conversely, many sophisticated design defects are often easy to find. The sophistication of the design mistake and the impact of the resulting defect are thus largely independent. Even trivial implementation errors can cause serious system problems. This is particularly important since the source of most software defects is simple programmer oversights and mistakes. While design issues are always important, initially developed programs typically have few design defects compared to the number of simple oversights, typos, and goofs. To improve program quality, it is thus essential that engineers learn to manage all the defects they inject in their programs.

The Engineer's Responsibility

The software engineer who writes a program is best able to find and fix its defects. It is thus important that software engineers take personal responsibility for the quality of the programs they produce. Writing defect-free programs, however, is challenging and it takes effective methods, skill, practice, and data. By using the Personal Software Process (PSP), engineers learn how to consistently produce essentially defect-free programs.

The importance of careful engineering practices is illustrated by the system test data for the Galileo spacecraft shown in Figure 1 [Nikora 91].

Figure 1. Galileo System Test Defects

In over six years of system testing, the Jet Propulsion Laboratory (JPL) found 196 defects in this 22,000 LOC system. While only 45 of these defects were judged to be fatal, large numbers of non-critical defects had to be removed before many of the critical defects could be uncovered. System testing is thus like peeling an onion; you need to remove the outer layers before you can find the critical problems. Since the Galileo spacecraft mission was successful, it is likely that they had removed almost all of the product's defects.

By following sound engineering practices, PSP-trained engineers remove almost all their defects before integration or system test. Systems built with these methods routinely have less than 0.2 defects per KLOC found in system test. When compared with the 8.9 defects/KLOC found in Galileo, it is clear that the PSP teaches engineers to peel away many layers of defects before they release their code.

The PSP and TSP

The Personal Software Process (PSP) was developed by the SEI to help small software groups and individual engineers understand and improve their capabilities. It provides a family of process scripts, forms, and standards that guide engineers through the steps of planning, tracking, and doing software work. The PSP is introduced with a textbook and course where engineers complete 10 programming exercises and 5 analysis reports [Humphrey 95]. The PSP is now being taught in several universities in the U.S., Europe, South America, and Australia, and it is being introduced by a growing number of software organizations.

We next developed the Team Software Process (TSP) because we found that many engineers had trouble applying the PSP to projects of more than one or two engineers. The TSP walks a team through launching a project and developing a product. While the TSP uses the four generic phases of requirements, design, implementation, and test, its emphasis is on multiple product versions and interleaved development activities. The TSP applies to teams of 2 to 20 hardware and software engineers who are PSP trained.

The TSP is introduced with a 3-day launch workshop where the engineers establish their goals, select their personal roles, and define their processes. They make a quality plan, identify support needs, and produce a detailed development schedule. The TSP guides them through a risk assessment, describes how to track and assess their work, and suggests ways to report their status to management. After a team has completed one 3-day launch workshop, a 2-day relaunch workshop is adequate for each subsequent phase. Then they integrate new team members, readjust role assignments, and reassess plans. The launch and relaunch workshops are not training courses; they are vital parts of the project.

The PSP and TSP have been used on development projects and have produced substantial cost, schedule, and quality improvements. In general, once they are PSP trained, engineers have one fifth to one tenth as many unit test defects as before and their integration, systems, acceptance, and usage defects are 20 to 100 or more times lower. One company, for example, found that PSP training reduced system test time from a norm of 2 to 3 months to 3 to 5 days. Since several references describe these results, I will not cover them further in this paper [Ferguson 97, Hayes 97, Humphrey 96].

PSP Quality Measurements

The principal process measures in software development are time, size, and defects. If one measures these precisely, then most other important measures can be derived. The way this is done in the PSP and TSP is as follows.

1. The development process is defined at several levels of detail.

2. At the lowest level, the process steps are reduced to individual engineering activities. Examples would be the detailed design for a program module, coding that module, or compiling the module until it compiles with no errors.
3. PSP-trained engineers track and record their activities for each process step. They track their time in minutes, count and record the defects they inject and remove, and measure the size product they produced.

While this level of detail might seem intrusive, the PSP shows engineers that such data are not difficult to gather. Engineers also find they need these data to plan their work and to improve their performance. The data help engineers understand their personal abilities and where and how they can improve. Consider the problem of a track team: without a measured track and a stopwatch, it would be hard to decide what events to enter or even how to practice. For the software engineer, the PSP provides the equivalent of a measured track and a stopwatch.

Finally, PSP data are the property of the engineers. While managers need data at a team level, they do not need to see the engineers' personal data. With the composite team data on time, size, and defects for every process phase, managers can analyze the quality of the process used and assess the defect content of the finished products.

A Software Quality Strategy

To help manage quality, we need measures of the defect content of our work. We also need to judge the number of defects remaining in the products we produce.

The PSP/TSP Quality Strategy

The PSP/TSP quality strategy is illustrated by the case of a large IBM program shown in Figure 2 [Kaplan 94].

Figure 2. Development vs. Usage Defects - IBM

The correlation between development and usage defects was 0.964 for release one. For release two, the correlation was also high at 0.878. The number of defects found in development test thus indicates the likely number of defects remaining after test. Therefore, to significantly reduce usage defects engineers must remove defects before development test. They can only do this by using a disciplined personal process. The PSP shows engineers how to remove defects at the earliest possible time, generally before the first compile.

TSP Quality Data

While several industrial groups are using the TSP, the first completed project was from a team at Embry Riddle Aeronautical University (ERAU). An overall quality measure showed that they removed 99.4% of the defects before system test. The team's defect removal profile is shown in Figure 3.

Figure 3. Defects/KLOC by Phase

The vertical scale on the left shows the defect density in defects per thousand lines of code (KLOC). The horizontal scale gives the development phases: detailed level design review (DLDR), code review (CDR), compile (COMP), unit test (UT), integration test (IT), and system test (ST).

In the PSP and TSP, the code review is done before any testing and even before the first compile. The engineers do these design and code reviews on their own code. After they have cleaned up their programs, peer reviews or inspections are then more effective at finding and removing more sophisticated problems. Testing will then be even more effective.

While the curve in Figure 3 looks superficially good, there were problems. An effective design review (DLDR) would have found at least two or three times as many defects as were found in unit test (UT). The curve in Figure 3 indicates that there were design review problems with at least some of the components.

The Quality Profile

By examining the large volume of PSP data, we can develop the profile of a process that should consistently produce high quality programs.

Some PSP Quality Data

We now have data on 2386 programs written in PSP courses. In all, the engineers took 15,534 hours to develop programs of 308,023 LOC. They found 22,644 defects. In Figure 4, we show how many defects were injected and removed per hour in each of the process phases. By examining the data on their personal work, engineers can estimate the numbers of defects they have injected and judge how long it would take them to remove these defects.

Figure 4. Defect Injection and Removal Rates

For example, engineers inject an average of 1.76 defects per hour in detailed design and remove 2.96 defects per hour in design review. Thus, for every hour of design, an engineer should plan on at least 0.59 hours of design review. Similarly, for every hour of coding, they would need an average of 0.64 hours of code review. From their personal data, engineers can thus estimate how long it should take to remove the defects. Note that these are average numbers and there is considerable variation among engineers.

By reviewing their defect data, engineers can see the benefits of careful practices. For example, while compiling, engineers inject an average of 0.60 defects per hour, while in test they inject an average of 0.38 defects per hour. Clearly, when engineers rush to get through compiling and testing, they make a lot of mistakes. When PSP-trained engineers start injecting large numbers of defects in compiling and testing, they know they need to stop and think or even to take a break. By watching their personal data, the PSP helps engineers better manage their work.

The PSP course data also show the value of spending an adequate amount of time in design. As shown in Figure 5, engineers have fewer test defects in their programs when they spend more time in design. Here, the bars at the front of the figure show the percentage of programs with various unit test defect densities. These front bars are for the case where the engineers spend enough design time and enough design review time.

Figure 5. Unit Test Defects vs. Practices

In this figure, the front bars are when engineers spent as much or more time in design as they spent in coding. For design review, they spent more than 50% of design time. The back bars, labeled Neither, are for those programs where engineers spent more time in coding than in designing and where they had an inadequate amount of design-review time. As you can see from the figure, product quality is worse with poor practices. For example, with good practices (the front bars), about 46% of the programs had no unit test defects. With poor practices, only about 18% had no defects. Most of the poor practice programs had 20 or more unit test defects/KLOC.

As you might expect, the situation is much the same for code reviews and compile defects. When code review times are greater than 50% of coding time, compile defects are lower. With adequate code review times, about 35% of the programs have 0 to 10 compile defects per KLOC and with little or no code review time, only 9% do. In this no-review case, the largest number of programs have over 100 defects/KLOC in compile.

Profile Values

The process limits in the quality profile are based on these PSP data. Design time should be greater than 100% of coding time, design review time should be greater than 50% of design time, and code review time should be greater than 50% of coding time. Also, compile defects should be less than 10/KLOC and unit test defects should be under 5/KLOC. When a factor meets or exceeds these criteria, that profile dimension is at the edge of the bullseye. When the criteria are not met, say 25% design review time instead of 50%, that dimension would be half way to the center of the bullseye. These criteria are summarized in Figure 6. Every organization should gather its own data and set its own profile values based on its own people, methods, tools, application areas, and experience.

Figure 6. Quality Profile Dimensions

Figure 7 shows an example of how these profiles are calculated. The data for this program component are as follows:

LOC: 336
Design time: 7.6 hours
Design review time (DLDR): 1.25 hours

Coding time: 8.9 hours
Code review time (CDR): 3.9 hours
Compile defects: 3
Unit test defects: 4

We next calculate the profile dimensions as follows:

Design/code time = 7.6/8.9 = 0.85 < 1.0,
so Design profile = 0.85/1.0 = 0.85
Design review/design time = 1.25/7.6 =
0.16 < 0.5, so
Design review profile = 0.16/0.5 = 0.33

Code review/code time = 3.9/8.9 = 0.44
< 0.5, so
Code review profile = 0.44/0.5 = 0.88

Unit test defects/KLOC = 1000*4/336 = 11.9 > 5.0, so
Unit test profile = 2*5.0/(11.9+5.0) = 0.61

Compile defects/KLOC = 1000*3/336 = 8.93 < 10.0, so
Compile profile = 1.0

While this component had three profile dimensions near 1.0, two were quite poor. Not surprisingly, this component had a defect found in integration testing.

Figure 7. Component E Quality Profile

Interpreting the Quality Profile

In reading the profiles, several guidelines can help to identify potentially troublesome components.

1. If engineers do not spend enough time during the design phase, they are probably designing while they are coding. As can be seen from Figure 4, they then introduce more than twice as many defects per hour as they would during a thorough design phase. The profile indicates this problem by a low value for design/code time.
2. Since engineers inject defects during design, they must spend enough time reviewing these designs or they will not find the defects. This problem is indicated by a low value for design review time.
3. Even when engineers spend enough time doing a design review, they may not do an effective review. Merely looking at the design for a long time, for example, will not generally result in finding many defects. Engineers must use sound review methods to find most of the sophisticated and even many of the simple design problems. Poor design-review methods are generally indicated by a high number of unit test defects/KLOC (a profile value near the center).
4. If engineers do not spend enough time reviewing their code before they compile it, they will likely leave defects to be found in later compiling and testing. The profile indicates this problem with a low value for code review time.
5. Even when engineers spend enough time doing a code review, they may not have done the review properly. This problem is generally indicated by a high number of compile defects/KLOC (a profile value near the center).

Note that profile values near the center indicate poor quality practices and suggest that the component will have defects found in the later testing or use.

A Quality Profile Analysis

The TSP is still in development but it has already been used by about 20 teams, including three student teams. The quality profiles for one student team at Embry Riddle Aeronautical University (ERAU) are shown in Figures 8 through 18. This team removed 99.4% of development defects before they started system test. Before reading the next paragraphs, see if you can identify the components that had defects in either system test or integration test or will likely have defects found in use. Note that several defects were found in integration test and system testing of this 5,090 LOC subsystem. In several months of experimental use, no further defects have been found.

Interpreting the Profiles

The way to read the ERAU quality profiles is as follows:

Component A (Figure 8). While this engineer did not do an adequate design or design review, she had a low number of unit test defects. Thus, while there is risk of later defects, it is not a high risk.

Figure 8. Component A Quality Profile

Component B (Figure 9). This engineer spent slightly less time than optimum in design, design review, and code review but the low numbers of compile and unit test defects indicate low risk.

Figure 9. Component B Quality Profile

Component C (Figure 10). Here, while design time was low and the unit test defects were slightly higher than desired, the design review time was adequate. This is a moderate risk component.

Figure 10. Component C Quality Profile

Component D (Figure 11). In this case, the engineers spent an inadequate amount of design review time and also found an excessive number of unit test defects. While code review time was low, the low compile defects indicate that there are likely few if any remaining coding errors. This component actually had an integration defect and likely has further undetected defects.

Figure 11. Component D Quality Profile

Component E (Figure 12). Component E also had low design review time and an excessive number of unit test defects. This component had an integration defect and could well have further problems. This is the component used in the earlier example in Figure 7.

Figure 12. Component E Quality Profile

Component F (Figure 13). This component is much like E only not quite as bad. It had one defect in system test.

Figure 13. Component F Quality Profile

Component G (Figure 14). This component looks reasonably good in all respects.

Figure 14. Component G Quality Profile

Component H (Figure 15). This component looks a bit risky. While adequate time was spent in both design review and code review, the excessive numbers of compile and unit test defects are a concern. This component did not have defects in integration or system test but it appears likely to have defects in later testing or use.

Figure 15. Component H Quality Profile

Component I (Figure 16). This component clearly has the worst of all the eleven profiles. While it had low unit test defects, it had an excessive number of compile defects and an inadequate amount of design and design review time. This component had an integration defect and seems likely to have more defects in the future.

Figure 16. Component I Quality Profile

Component J (Figure 17). The principal risk here is that there were an excessive number of unit test defects. While this component had no integration or system test defects, it is exposed.

Figure 17. Component J Quality Profile

Component K (Figure 18). This is another good looking profile. This component had no integration or system test defects.

Figure 18. Component K Quality Profile

Conclusions

With TSP data, engineers can determine which components are most likely to have defects before they even start integration and system testing. By thoroughly inspecting and reworking these defect-prone components before integrating or system testing them, development teams can save a substantial amount of test time and produce higher quality products.

These results can best be understood by considering some data. First, before PSP training, most experienced engineers inject between 80 and 125 defects per KLOC. They will then find about half of the remaining defects in each subsequent compile and test phase. Thus, with the Galileo spacecraft, if we assume that JPL used very good engineers who injected only 80 defects per KLOC, that would be 1760 defects, of which about 880 would be found in compile, about 440 in unit testing, and about 220 in integration test. That would leave about 220 defects at system test entry. In over six years of testing, they found 196 of these defects. Even though the mission was a success, it thus seems likely that the product still had a few remaining defects. To achieve this level of quality for a small 22,000 LOC product, however, JPL had to test for over six years.

With the PSP, engineers remove almost all their defects before integration or system test. Through a combination of sound design practices and careful reviews, they consistently produce products with fewer than 0.2 system-test defects/KLOC and essentially none in later use. With PSP training, the JPL engineers would then have had 4 or 5 defects to find in system test and they could have completed testing in a few weeks instead of six years.

When engineers know how to measure their work, they can obtain a great deal of useful quality information. The quality profile is one example of what one can learn about product quality, even before integration test entry. When engineers gather and use process data, they can manage product quality, save test time, cut development costs, and shorten schedules.

Acknowledgements

Several people have kindly reviewed the draft of this paper and made helpful suggestions. I particularly thank Alan Koch, Mark Paulk, and Bill Peterson. I also thank the anonymous journal reviewers. Their perceptive comments were most helpful. Dr. Iraj Hirmanpoor and Dr. Soheil Khajenoori and the student team at Embry Riddle Aeronautical University also deserve special thanks. They have provided strong support and a useful test environment for the early PSP and TSP work.

References

[Ferguson 97] Pat Ferguson, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya, "Introducing the Personal Software Process: Three Industry Case Studies," *IEEE Computer*, vol. 30, no. 5, pp 24-31, May 1997.

[Hayes 97] Will Hayes, "The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers," CMU/SEI-97-TR-001

[Humphrey 95] W. S. Humphrey, *A Discipline for Software Engineering.* Reading, MA: Addison-Wesley, 1995.

[Humphrey 96] W. S. Humphrey, "Using a Defined and Measured Personal Software Process," *IEEE Software*, May, 1996.

[Humphrey 97] W. S. Humphrey, *Introduction to the Personal Software Process.* Reading, MA: Addison-Wesley, 1997.

[Kaplan 94] Craig Kaplan, Ralph Clark, and Victor Tang, *Secrets of Software Quality, 40 Innovations from IBM*. New York, N.Y.: McGraw-Hill, Inc., 1994.

[Leveson 95] Nancy G. Leveson, *Safeware, System Safety and Computers*. Reading, MA: Addison Wesley, 1995.

[Nikora 91] Allen P. Nikora, "Error Discovery Rate by Severity Category and Time to Repair Software Failures for Three JPL Flight Projects," Software Product Assurance Section, Jet Propulsion Laboratory 4800 Oak Grove Drive, Pasadena, CA 91109-8099, November 5, 1991.

[Perrow 84] Charles Perrow, *Normal accidents, Living with High-Risk Technologies*. New York, NY: Basic Books, Inc., 1984.

For More Information

For more information about the Software Engineering Institute, please contact

Customer Relations
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Phone, Voicemail, and On-Demand FAX: 412-268-5800
E-mail: customer-relations@sei.cmu.edu

Software Engineering Institute | Carnegie Mellon

The Software Quality Profile (Figures)

Figure 1. Galileo System Test Defects

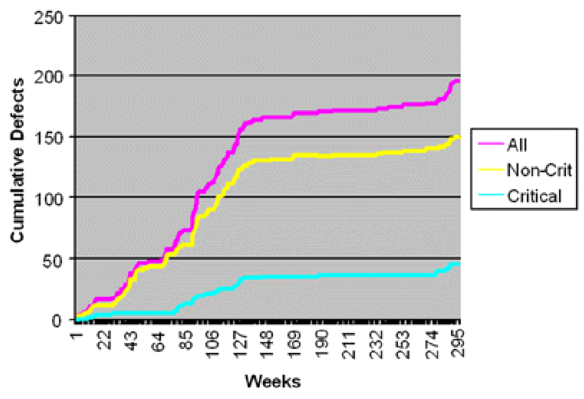


Figure 2. Development vs. Usage Defects - IBM Release 1 (r = 0.9644)

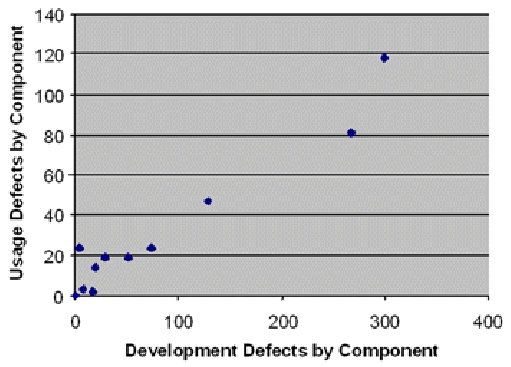


Figure 3. Defects/KLOC by Phase

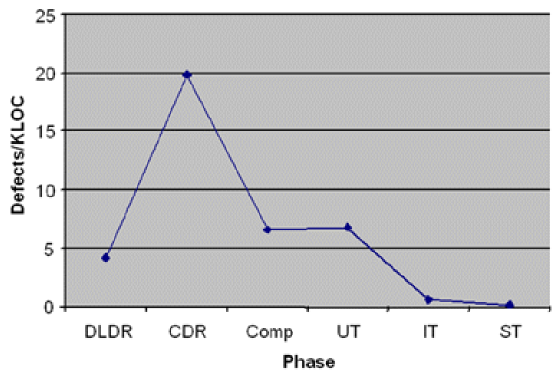


Figure 4. Defect Injection and Removal Rates

PSP Phase	Injected/hour	Removed/hour	Removed/Injected
DLD	1.76	0.10	0.05
DLDR	0.11	2.96	27.91
Code	4.20	0.51	0.10
CDR	0.11	6.52	59.78
Compile	0.60	9.48	15.84
Test	0.38	2.21	5.82

Figure 5. Unit Test Defects vs. Practices

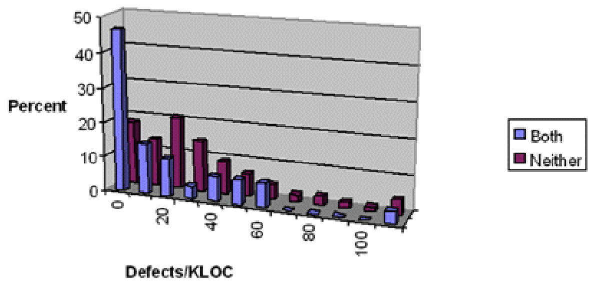


Figure 6. Quality Profile Dimensions

Dimension	Meaning
Design/Code Time	The ratio of detailed design to coding time - when engineers do not take the time to produce a thorough design, they generally make more design errors. To reduce this risk, design time should equal at least 100% of coding time.

Code Review Time	The time spent in code review, compared with coding time - by doing a personal code review before they compile, engineering can find a large percentage of their defects. A thorough code review should take 50% or more of coding time.
Compile Defects/KLOC	The defects per KLOC found in compile - even with good review times and rates, the review still could have missed a lot of defects. For quality products, compile defects should be less than 10 defects/KLOC.
Design Review Time	Detailed design review time, related to detailed design time - a thorough detailed design review should take 50% or more of the time spent in detailed design. Anything less generally indicates an inadequate review.
Unit Test Defects/KLOC	The defects per KLOC found in unit test - the number of defects found in unit test is one of the best indicators of the number that will later be found. When the unit test defects/KLOC exceed 5, subsequent problems are likely.

Figure 7. Component E Quality Profile

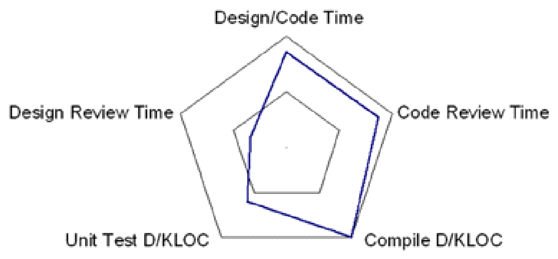


Figure 8. Component A Quality Profile

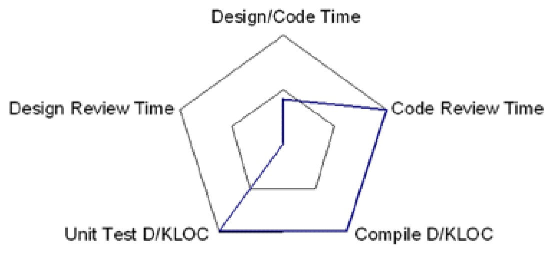


Figure 9. Component B Quality Profile

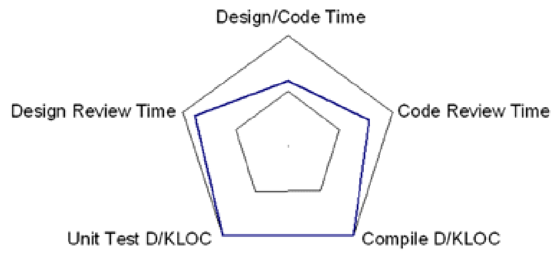


Figure 10. Component C Quality Profile

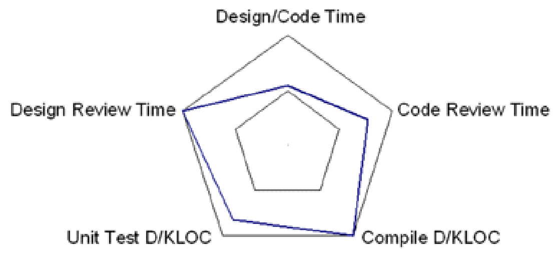


Figure 11. Component D Quality Profile

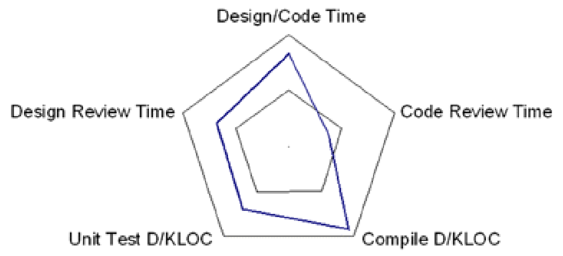


Figure 12. Component E Quality Profile

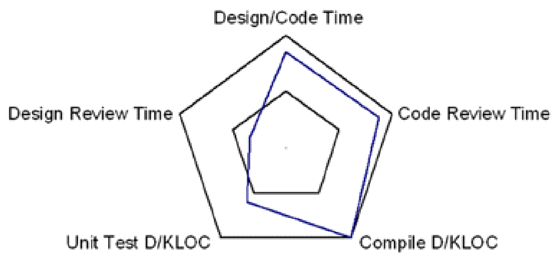


Figure 13. Component F Quality Profile

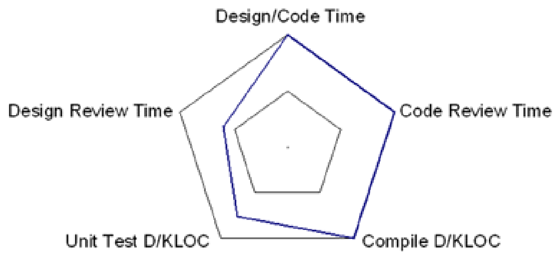


Figure 14. Component G Quality Profile

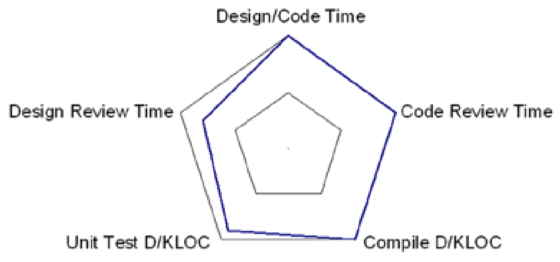


Figure 15. Component H Quality Profile

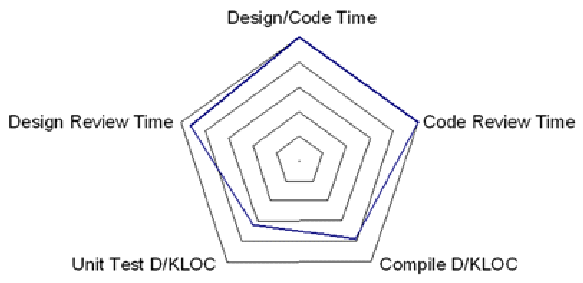


Figure 16. Component I Quality Profile

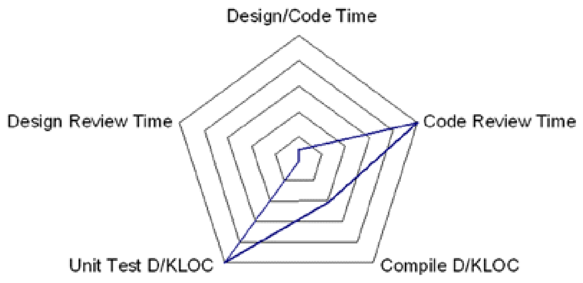


Figure 17. Component J Quality Profile

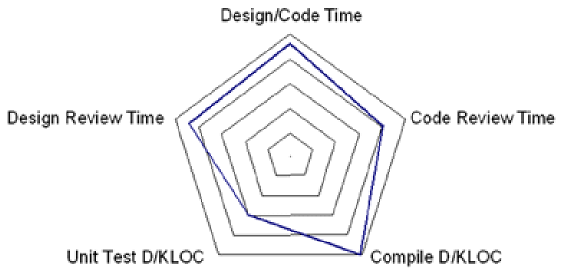
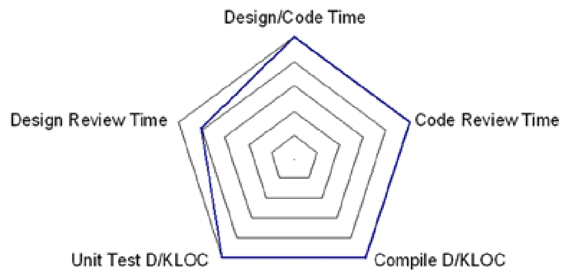


Figure 18. Component K Quality Profile


Design/Code Time
Code Review Time
Compile D/KLOC
Unit Test D/KLOC
Design Review Time