MEASURING DEFECT POTENTIALS

AND DEFECT REMOVAL EFFICIENCY

Version 4.1– March 1, 2008

Provided by Rex Black Consulting Services (www.rbcs-us.com).

Abstract

There are two measures that have a strong influence on the outcomes of software projects: 1) Defect potentials; 2) Defect removal efficiency.

The term "defect potentials" refers to the total quantity of bugs or defects that will be found in five software artifacts: requirements, design, code, documents, and "bad fixes" or secondary defects.

The term defect removal efficiency refers to the percentage of total defects found and removed before software applications are delivered to customers.

As of 2008 the U.S. average for defect potentials is about 5 defects per function points. The U.S. average for defect removal efficiency is only about 85%. The U.S. average for delivered defects is about 0.75 defects per function point.

Software project costs and schedules decline for projects whose cumulative defect removal efficiency level is approximately 95%. Achieving 95% removal efficiency requires a combination of formal inspections and formal testing. Testing alone is insufficient for optimal defect removal efficiency.

Cautions are given against measuring quality using either "cost per defect" or "defects per KLOC' because both metrics violate standard economics and yield incorrect results.

Capers Jones, Founder and Chief Scientist Emeritus
Software Productivity Research, LLC

Email cjonesiii@CS.com

Measuring Defect Potentials and Defect Removal Efficiency

Introduction

There are two very important measurements of software quality that are critical to the industry:

1) Defect potentials
2) Defect removal efficiency

All software managers and quality assurance personnel should be familiar with these measurements, because they have the largest impact on software quality and also software costs and schedules of any known measures.

Measuring Defect Potentials

The phrase "defect potentials" refers to the probable numbers of defects that will be found during the development of software applications. As of 2008 the approximate U.S. averages for defects in these five categories follow, measured in terms of defects per function point and rounded slightly so that the cumulative results are an integer value for consistency with other publications by the author.

Note that defect potentials should be measured with function points and not with lines of code. This is because most of the serious defects are not found in the code itself, but rather in requirements and design. Following are U.S. averages for defect potentials circa 2008 using defects per function point. (Version 4.2 of the International Function Point Users Group counting rules are assumed.)

Requirements defects	1.00
Design defects	1.25
Coding defects	1.75
Documentation defects	0.60
Bad fixes	0.40
Total	5.00

The measured range of defect potentials ranges from just below 2.00 defects per function point to about 10.00 defects per function point. Defect potentials correlate with application size. As application sizes increase, defect potentials also rise. (Defect potentials also vary with the type of software, with CMM levels, and in response to other factors.)

A useful approximation of the relationship between defect potentials and defect size is a simple rule of thumb: application function points raised to the 1.25 power will yield the approximate defect potential for software applications. Actually, this rule applies primarily to applications developed by organizations at CMM level 1. For the higher CMM levels, lower powers would result. Reference 8 shows additional factors that affect the rule of thumb.

(Note that the averages for defect potentials are derived from studies of about 600 companies and 13,000 projects. Non-disclosure agreements prevent the identification of most specific companies. However some companies such as IBM and ITT have provided data on defect potentials and removal efficiency levels.)

A corollary rule of thumb can predict the probable number of test cases that will be needed to fully test software projects. Raising the function point total of the application to the 1.2 power will give a useful approximation of total test cases for all forms of testing.

The phrase "defect removal efficiency" refers to the percentage of the defect potentials that will be removed before the software application is delivered to its users or customers. As of 2008 the U.S. average for defect removal efficiency is about 85%.

If the average defect potential is 5.00 bugs or defects per function point and removal efficiency is 85%, then the total number of delivered defects will be about 0.75 defects per function point. However, some forms of defects are harder to find and remove than others. For example requirements defects and bad fixes are much more difficult to find and eliminate than coding defects as will be discussed later in the article.

Cautions about Hazardous Quality Metrics

Strong cautions need to be given about two common metrics that are often used for quality, but which invariably yield inaccurate results: 1) cost per defect; 2) defects per KLOC.

The "cost per defect" metric violates standard economics because it always goes up where the lowest numbers of defects are found, and goes down where the largest numbers of defects are found. The reason for this is because defects can be reduced faster than defect removal personnel can be reassigned.

Assume a software tester is paid $5000 per month. When working on a buggy project the tester finds 100 defects in the course of the first month of testing. The cost per defect would be $50. But suppose that in the second month of testing only 25 defects are found. Now the cost has risen to $200 per defect. As can be seen, cost per defect rises as discovered defects go down. Suppose for the third month only five defects are found but the tester's monthly salary of $5000 still has to be paid. Now we have a cost of $1000 per defect. In the fourth month if no defects at all were found, we would have an infinite cost per defect! These simple examples ignore many factors, but illustrate the point that "cost per defect" essentially ignores fixed costs. Also, you cannot add or average costs per defect and match any form of standard economics.

Measuring "defect removal cost per function point" is more reliable and matches standard economic assumptions better than either cost per defect or cost per KLOC. For example if the application being tested were 2,500 function points in size then the first month's testing cost, assuming the same tester and the same $5000 monthly salary, would be $2.00 per function point, as would the costs for months two, three, and four. At the end of testing the four months costs can be summed to produce a cost of $8.00 per function point for the tester in question. Other costs such as test case creation can also be measured in the same fashion.

Time and motion studies of actual defect repairs shows that fixing most bugs in code requires about four hours, regardless of whether the bug is found before release or afterwards. Shipping the repairs to customers does cost more, but with electronic communications not a lot more. There are two forms of defect repair that are much more expensive than average: 1) abeyant defects; 2) repairs to "error-prone" modules with high complexity levels.

The term "abeyant defect" refers to a reported bug where the maintenance or repair team cannot duplicate the error and therefore has no immediate way to fix the problem. Usually some special combination of hardware and software at the client site is the cause of abeyant defects. It may be necessary to visit the site, or at any rate to gather much more data than a simple bug report. IBM data suggests that abeyant defects comprise about 5% of customer-reported errors.

The phrase "error-prone module" refers to a small percentage of modules in large applications which receive a disproportionate number of defect reports. As a rule of thumb, about 5% of the modules in large applications will receive as many as 50% of all reported defects. Error-prone modules were discovered by IBM in the 1970's, and have been found to be common in the software industry. Usually error-prone modules are so high in complexity that they are hard to fix. Worse, the bad-fix injection rate or secondary defects accidentally included in bug fixes can top 25%; i.e. one out of every four bug repairs contains a fresh bug. If error-prone modules are present in large applications, the only effective solution is surgical removal. They cannot usually be fixed via minor updates.

The "KLOC metric" or defects per 1000 lines of code has four problems. First, requirements and design defects outnumber coding defects. Second, there are more than 700 programming languages in existence and for many of these there are no effective rules for counting lines of code. Third, many applications use multiple programming languages and code counting is very difficult when more than two languages are used in the same software application. (Some applications use more than 12 programming languages.) Fourth, and most serious, lines of code metrics tend to penalize modern object-oriented and high-level programming languages and make older low-level languages look better than they are. This last point needs an example.

Assume an application requires 10,000 lines of assembly code (10 KLOC) and has 100 requirements and design bugs and 200 coding bugs, for a total of 300. The defects per KLOC would be 30. Assuming this application is 40 function points in size, the total defects would be 7.5 per function point.

Now assume the same software is written in Java and requires only 2,000 lines of code (2 KLOC). There are still 100 requirement and design bugs, but coding bugs have dropped to 50. Now the total defects are only 150 for a 50% reduction. However defects per KLOC have risen to 75. The application still contains 40 function points, so the number of defects per function point is only 3.75. When non-code defects are included in the count (as they should be) then “defects per KLOC” will penalize high-level programming languages because coding defects decline, but requirements and design defects are independent of the code and stay relatively constant for all programming languages.

Even when only coding bugs are measured, KLOC metrics still violate standard economic assumptions. The assembly language version contained 200 coding bugs or 20 per KLOC. The Java version contained only 50 bugs, for a reduction of 150 coding bugs. But when measured in terms of defects per KLOC, the Java example has 25 bugs per KLOC and so looks worst than the assembly-language version.

However when measured using function point metrics, the assembly code bugs total 5 per function point and the Java code bugs total only 1.25 bugs per function point. As can be seen, the use of function point metrics correctly matches the reduced volume of code bugs due to the higher-level programming language.

The “cost per defect” metric penalizes quality and achieves the lowest cost per defect for the buggiest applications. The “defects per KLOC” metric penalizes high-level programming languages and achieves the lowest cost per KLOC for basic assembly language. Therefore these two metrics have actually concealed and distorted the economic value of quality for more than 50 years.

Incidentally some of the new and specialized metrics are also inadequate for studies of quality economics. For example story points, use case points, web-object points, and some of the object-oriented metrics such as MOOSE are so narrow in focus and so specialized that they cannot be used for broad benchmark studies that include other kinds of software.

Returning to the main subject, at a more granular level, the U.S. averages for defect removal efficiency against each of the five defect categories is approximately the following:

Defect Origin	Defect Potential	Removal Efficiency	Defects Remaining
Requirements defects	1.00	77%	0.23
Design defects	1.25	85%	0.19
Coding defects	1.75	95%	0.09
Documentation defects	0.60	80%	0.12
Bad fixes	0.40	70%	0.12
Total	5.00	85%	0.75

Note that the defects discussed in this section include all severity levels, ranging from severity 1 “show stoppers” down to severity 4 “cosmetic errors.” Obviously it is important to measure defect severity levels as well as recording numbers of defects.

(The normal period for measuring defect removal efficiency starts with requirements inspections and ends 90 days after delivery of the software to its users or customers. Of course there are still latent defects in the software that won't be found in 90 days, but having a 90-day interval provides a standard benchmark for defect removal efficiency. It might be thought that extending the period from 90 days to six months or 12 months would provide more accurate results. However, updates and new releases usually come out after 90 days, so these would dilute the original defect counts.

Latent defects found after the 90 day period can exist for years, but on average about 50% of residual latent defects are found each calendar year. The results vary with number of users of the applications. The more users, the faster residual latent defects are discovered. Results also vary with the nature of the software itself. Military, embedded, and systems software tends to find bugs or defect more quickly than information systems.)

Defect potentials vary with application size, and they also vary with the class and type of software. Table 1 shows the approximate average defect potentials noted for six types of software projects:

Table 1: Average Defect Potentials for Six Application Types
(Data expressed in terms of "defects per function point"

	Web	MIS	Outsource	Commercial	System	Military	Average
Requirements	1.00	1.00	1.10	1.25	1.30	1.70	1.23
Design	1.00	1.25	1.20	1.30	1.50	1.75	1.33
Code	1.25	1.75	1.70	1.75	1.80	1.75	1.67
Documents	0.30	0.60	0.50	0.70	0.70	1.20	0.67
Bad Fix	0.45	0.40	0.30	0.50	0.70	0.60	0.49
TOTAL	4.00	5.00	4.80	5.50	6.00	7.00	5.38

The form of defect called "bad fix" in table 1 is that of secondary defects accidentally present in a bug or defect repair itself. In one lawsuit where the author worked as an expert witness, data recorded by the plaintiff showed that the defendant made four consecutive attempts to fix a bug, and each attempted repair contained a new bug. Each of the four attempts was claimed to fix the problem, but not until the fifth attempt was the original problem actually fixed, and no new problems were introduced.

There are large ranges in terms of both defect potentials and defect removal efficiency levels. The "best in class" organizations have defect potentials that are below 2.50 defects per function point coupled with defect removal efficiencies that top 95% across the board. Projects that are below 3.0 defects per function point coupled with a cumulative defect removal efficiency level of about 95% tend to be lower in cost and shorter in development schedules than applications with higher defect potentials and lower levels of removal efficiency.

Observations of projects that run late and have significant cost overruns show that the primary cause of these problems are excessive quantities of defects that are not discovered nor removed until testing starts. Such projects appear to be on schedule and within budget until testing begins. Delays and cost overruns occur when testing starts, and hundreds or even thousands of latent defects are discovered. The primary schedule delays occur due to test schedules far exceeding their original plans.

Defect removal efficiency levels peak at about 99.5%. In examining data from about 13,000 software projects over a period of 40 years, only two projects had zero defect reports in the first year after release. This is not to say that achieving a defect removal efficiency level of 100% is impossible, but it is certainly very rare.

Organizations with defect potentials higher than 7.00 per function point coupled with defect removal efficiency levels of 75% or less can be viewed as exhibiting professional malpractice. In other words, their defect prevention and defect removal methods are below acceptable levels for professional software organizations.

Measuring Defect Removal Efficiency

Most forms of testing average only about 30% to 35% in defect removal efficiency levels and seldom top 50%. Formal design and code inspections, on the other hand, often top 85% in defect removal efficiency and average about 65%. With every form of defect removal having a comparatively low level of removal efficiency, it is obvious that many separate forms of defect removal need to be carried out in sequence to achieve a high level of cumulative defect removal. The phrase "cumulative defect removal" refers to the total number of defects found before the software is delivered to its customers.

Table 2 shows patterns of defect prevention and defect removal for the same six forms of software shown in Table 1:

Table 2: Patterns of Defect Prevention and Removal Activities

	Web	MIS	Outsource	Commercial	System	Military
Prevention Activities						
Prototypes	20.00%	20.00%	20.00%	20.00%	20.00%	20.00%
Six Sigma					20.00%	20.00%
JAD sessions		30.00%	30.00%			
QFD sessions					25.00%	
Subtotal	*20.00%*	*44.00%*	*44.00%*	*20.00%*	*52.00%*	*36.00%*
Pretest Removal						
Desk checking	15.00%	15.00%	15.00%	15.00%	15.00%	15.00%
Requirements review			30.00%	25.00%	20.00%	20.00%
Design review			40.00%	45.00%	45.00%	30.00%
Document review				20.00%	20.00%	20.00%
Code inspections				50.00%	60.00%	40.00%
Ind. Verif. and Valid.						20.00%
Correctness proofs						10.00%
Usability labs				25.00%		
Subtotal	*15.00%*	*15.00%*	*64.30%*	*89.48%*	*88.03%*	*83.55%*
Testing Activities						
Unit test	30.00%	25.00%	25.00%	25.00%	25.00%	25.00%
New function test		30.00%	30.00%	30.00%	30.00%	30.00%
Regression test			20.00%	20.00%	20.00%	20.00%
Integration test		30.00%	30.00%	30.00%	30.00%	30.00%
Performance test				15.00%	15.00%	20.00%
System test		35.00%	35.00%	35.00%	40.00%	35.00%
Independent test						15.00%
Field test				50.00%	35.00%	30.00%
Acceptance test			25.00%		25.00%	30.00%
Subtotal	*30.00%*	*76.11%*	*80.89%*	*91.88%*	*92.69%*	*93.63%*
Overall Efficiency	52.40%	88.63%	96.18%	99.32%	99.58%	99.33%

Some forms of defect removal such as desk checking and unit testing are normally performed privately by individuals and are not usually measured. However several companies such as IBM and ITT have collected data on these methods from volunteers, who agreed to record "private" defect removal activities in order to judge their relative defect removal efficiency levels. (The author himself was once such a volunteer, and was troubled to note that his private defect removal activities were less than 35% efficient.)

Several new approaches are not illustrated by either table 1 or table 2 but have been discussed elsewhere in other books and articles. Watts Humphrey's Team Software Process (TSP) and Personal Software Process (PSP) are effective in both defect prevention and defect removal context. The five levels of the SEI capability maturity model (CMM) and the newer capability maturity model integrated (CMMI) are also effective in improving both defect potentials and defect removal efficiency levels. Some of the new automated testing tools and methods tend to increase the defect removal efficiency levels of the tests they support, but such data is new and difficult to extract from overall test results. The Agile software development method has proven to be effective in lowering defect potentials for applications below 2,500 function points in size. There is not yet enough data for larger sizes of projects, or for defect removal, to be certain about the effects of the Agile method for larger software applications.

There are some new approaches that make testing more difficult rather than easier. For example some application generators that produce source code directly from specifications are hard to test because the code itself is not created by humans. Formal inspections of the original specifications are needed to remove latent defects. It would be possible to use sophisticated text analysis programs to "test" the specifications, but this technology is outside the domain of both software quality assurance (SQA) and normal testing organizations. There is very little data available on defect potentials and defect removal efficiency levels associated with application generators.

As can be seen from the short discussions here, measuring defect potentials and defect removal efficiency provide the most effective known ways of evaluating various aspects of software quality control. In general, improving software quality requires two important kinds of process improvement: 1) defect prevention; 2) defect removal.

The phrase "defect prevention" refers to technologies and methodologies that can lower defect potentials or reduce the numbers of bugs that must be eliminated. Examples of defect prevention methods include joint application design (JAD), structured design, and also participation in formal inspections. (Formal design and code inspections are the most effective defect removal activity in the history of software, and are also very good in terms of defect prevention. Once participants in inspections observe various kinds of defects in the materials being inspected, they tend to avoid those defects in their own work. All software projects larger than 1000 function points should use formal design and code inspections.)

The phrase "defect removal" refers to methods that can either raise the efficiency levels of specific forms of testing, or raise the overall cumulative removal efficiency by adding additional kinds of review or test activity. Of course both approaches are possible at the same time.

In order to achieve a cumulative defect removal efficiency of 95%, it will be necessary to use approximately the following sequence of at least eight defect removal activities:

1. Design inspections
2. Code inspections
3. Unit test
4. New function test
5. Regression test
6. Performance test
7. System test
8. External Beta test

To go above 95%, additional removal stages will be needed. For example requirements inspections, test case inspections, and specialized forms of testing such as human factors testing, performance testing, and security testing add to defect removal efficiency levels.

Since each testing stage will only be about 30% efficient on average, it is not feasible to achieve defect removal efficiency level of 95% by means of testing alone. Formal inspections before testing will not only remove most of the defects before testing begins, they will also raise the efficiency level of each test stage. Inspections benefit testing because design inspections provide a more complete and accurate set of specification from which to construct test cases. Code inspections are beneficial because the human mind can find problems that often are elusive, such as requirements defects that find their way into the code.

Formal design and code inspections prior to testing have been observed to raise the efficiency of subsequent test stages by around 5%. Also, formal inspections are very effective in terms of defect prevention. Participants in formal inspections tend to avoid making the same kinds of problems that are found during the inspection process.

From an economic standpoint, combining formal inspections and formal testing will be cheaper than testing by itself. Inspections and testing in concert will also yield shorter development schedules than testing alone. This is because when testing starts after inspections, almost 85% of the defects will already be gone. Therefore testing schedules will be shortened by more than 45%.

When IBM applied formal inspections to a large data base project, it was interesting that delivered defects were reduced by more than 50% from previous releases. The overall schedule was shortened by about 15%. Testing itself was reduced from two shifts over a 60 day period to one shift over a 40 day period. More important, customer satisfaction improved to "good" from prior releases, where customer satisfaction had been "very poor". Cumulative defect removal efficiency was raised from about 80% to just over 95% as a result of using formal design and code inspections.

Measurement of Both Defect Potentials and Defect Removal Efficiency

Measuring defect potentials and defect removal efficiency levels are among the easiest forms of software measurement, and are also the most important. To measure defect potentials it is necessary to keep accurate records of all defects found during the development cycle, which is something that should be done as a matter of course. The only difficulty is that "private" forms of defect removal such as unit testing will need to be done on a volunteer basis.

Measuring the numbers of defects found during reviews, inspections, and testing is also straight-forward. To complete the calculations for defect removal efficiency, customer-reported defect reports submitted during a fixed time period are compared against the internal defects found by the development team. The normal time period for calculating defect removal efficiency is 90 days after release.

As an example, if the development and testing teams found 900 defects before release, and customers reported 100 defects in the first three months of usage, it is apparent that the defect removal efficiency would be 90%.

Although measurements of defect potentials and defect removal efficiency levels should be carried out by 100% of software organizations, unfortunately the frequency of these measurements circa 2008 is only about 5% of U.S. companies. In fact more than 50% of U.S. companies don't have any useful quality metrics at all. More than 80% of U.S. companies, including the great majority of commercial software vendors, have only marginal quality control and are much lower than the optimal 95% defect removal efficiency level. This fact is one of the reasons why so many software projects fail completely, or experience massive cost and schedule overruns. Usually failing projects seem to be ahead of schedule until testing starts, at which point huge volumes of unanticipated defects stop progress almost completely.

As it happens, projects that average about 95% in cumulative defect removal efficiency tend to be optimal in several respects. They have the shortest development schedules, the lowest development costs, the highest levels of customer satisfaction, and the highest levels of team morale. This is why measures of defect potentials and defect removal efficiency levels are important to the industry as a whole. These measures have the greatest impact on software performance of any known metrics.

From an economic stand point, going from the U.S. average of 85% in defect removal efficiency up to 95% actually saves money and shortens development schedules. This is because most schedule delays and cost overruns are due to excessive defect volumes during testing. However, to climb above 95% defect removal efficiency up to 99% does require additional costs. It will be necessary to perform 100% inspections of every deliverable, and testing will require about 20% more test cases than normal.

It is an interesting sociological observation that measurements tend to change human behavior. Therefore it is important to select measurements that will cause behavioral changes in positive and beneficial directions. Measuring defect potentials and defect removal efficiency levels have been noted to make very beneficial improvements in software development practices.

When these measures were introduced into large corporations such as IBM and ITT, in less than four years the volumes of delivered defects had declined by more than 50%; maintenance costs were reduced by more than 40%; development schedules were shortened by more than 15%. There are no other measurements that can yield such positive benefits in such a short time span. Both customer satisfaction and employee morale improved too, as a direct result of the reduction in defect potentials and the increase in defect removal efficiency levels.

Although this article concentrates on improvements in software development, longer range studies that include from four to 10 years of software maintenance indicate that quality improvements benefit software economics throughout the lifecycle. In fact, total cost of ownership (TCO) can be improved by more than 50% from a combination of low initial defect potentials and high levels of defect removal efficiency.

The Economics of Defect Prevention and Defect Removal

Due to many years of unreliable economic studies based on "cost per defect" and "KLOC" metrics, the economic advantages of low defect potentials and high levels of defect removal efficiency are not well covered in the software quality literature. This section illustrates the advantages of quality by showing two case studies. Both cases are assumed to be systems software of 2,500 function points in size. Both cases assume a fully burdened salary rate of $10,000 per staff month.

Case 1 illustrates a somewhat worse than average project of 2,500 function points in size that used only testing for defect removal. Case 2 illustrates a project that used formal design and code inspections before testing, plus one additional testing step. The data in both cases is derived from actual projects, but simplified to illustrate basic points.

Case 1: Low Quality Results

Case 1 is assumed to be a fairly small systems software project of 2,500 function points in size. The development team is assumed to be level 1 on the CMM scale. Defect potentials are derived by raising the function point total of the application to the 1.25 power, which results in a total of 17,678 defects or 7.07 defects per function point. Defect removal efficiency is assumed to be 85% overall. Defect removal operations consist of six test stages: 1) unit test, 2) new function test, 3) regression test, 4) integration test, 5) system test, and 6) external Beta test.

The Case 1 development schedule was approximately 23 calendar months. Development productivity was about 7 function points per staff month. Total development effort was 357 staff months. Development costs were approximately $3,500,000 or $1,400 per function point. The cost of quality or total dollars spent on defect removal amounted to about 35% or $1,250,000, which is equal to $500 per function point.

With a defect removal efficiency of 85%, a total of just over 1,500 defects would be found, leaving more than 2,650 to be delivered to customers. Of these about 25% or more than 660 would be high-severity defects equivalent to severity 1 and severity 2 on most severity scales.

Case 2: High Quality Results

Case 2 is exactly the same size and the same class of software as Case 1. The development team is assumed to be level 3 on the CMM scale. By means of more effective defect prevention such as Quality Function Deployment (QFD) and Six-Sigma the defect potentials are lower. Raising function point totals to the 1.2 power would yield a potential of about 11,954 defects or 4.78 defects per function point. Defect removal efficiency is assumed to be 95%. Defect removal operations consist of nine stages: 1) design inspections; 2) code inspections; 3) unit test, 4) new function test, 5) regression test, 6) integration test, 7) performance test, 8) system test, 9) external Beta test.

The Case 2 development schedule was approximately 20 calendar months, or three months shorter than Case 1. Productivity was about 10 function points per staff month. Most of the schedule savings are due to shorter testing cycles, which in turn are due to lower defect potentials plus pre-test inspections. Development productivity for Case 2 was about 10 function points per staff month. Total development effort was 250 staff months. Development costs were approximately \$2,500,000 or \$1,000 per function point. The cost of quality or total dollars spent on defect removal amounted to about 33% or \$825,000, which is equal to \$330 per function point.

With a defect removal efficiency of 95%, a total of just over 11,350 defects were found leaving about 600 to be delivered to customers. Of these about 15% or 90 would be high-severity defects equivalent to severity 1 and severity 2 on most severity scales.

To clarify the differences between the two case studies, Table 3 presents a side-by-side comparison. Note that both examples are exactly the same size, but differ in these key elements:

- CMM levels
- Defect prevention
- Defect potentials
- Defect removal efficiency
- Development schedules
- Development effort
- Development costs

Table 3 summarizes these differences:

Table 3: Comparison of High and Low Quality Results for Identical Applications (Data is expressed in terms of IFPUG function points, version 4.2)

	Case 1 Low Quality	**Case 2 High Quality**	**Difference**
Function Point Size	2,500	2,500	0
Monthly cost	$10,000	$10,000	0
CMM Level	1	3	2
Schedule months	22.87	19.55	-3.31
Effort (staff months)	357	250	-107
Development Costs	$3,571,429	$2,500,000	-$1,071,429
Cost per Function Pt.	$1,429	$1,000	-$429
Cost of Quality (COQ)	$1,250,000	$825,000	-$425,000
COQ Cost per Function Pt.	$500	$330	-$170
Defect potentials	17,678	11,954	-5,723
Per Function Point	7.07	4.78	-2.29
Defect Removal	85.00%	95.00%	10.00%
Defects removed	15,026	11,357	-3,669
Defects delivered	2,652	598	-2,054
Per Function Point	1.06	0.24	-0.82
High-severity defects	663	90	-573
Per Function Point	0.27	0.04	0.23
Defect removal stages	6	9	3

Because defect potentials are reduced at the same time that defect removal efficiency has gone up, the overall results between the two cases are quite significant. Note that high-severity defects declined to only 13.6% of the low quality example. Such a reduction in serious defects is a major factor associated with customer satisfaction.

In real life as well as in these two examples, lowering defect potentials and raising defect removal efficiency levels up to 95% will benefit quality, schedules, costs, user satisfaction, and team morale at the same time.

Phil Crosby, the former ITT Vice President of Quality wrote an important book entitled "Quality is Free" (reference 2). For software projects, quality is not only free, it reduces both development and maintenance costs at the same time and tremendously improves customer satisfaction.

References

Note: This article uses excerpts from the author's books Estimating Software Costs (McGraw Hill 2007 and Applied Software Measurement (McGraw Hill 2008).

1) Boehm, Barry W.; Software Engineering Economics; Prentice Hall, Englewood Cliffs, NJ; 1981; 767 pages.

2) Crosby, Philip B.; Quality is Free; New American Library, Mentor Books, New York, NY; 1979; 270 pages.

3) Garmus, David & Herron, David; Function Point Analysis; Addison Wesley Longman, Boston, MA; 2001; ISBN 0-201-69944-3; 363 pages.

4) Garmus, David & Herron, David; Measuring the Software Process: A Practical Guide to Functional Measurement; Prentice Hall, Englewood Cliffs, NJ; 1995.

5) Grady, Robert B. and Caswell, Deborah L.; Software Metrics: Establishing a Company Wide Program; Prentice-Hall, Inc.; 1987; 288 pages.

6) International Function Point Users Group; IT Measurement; Addison Wesley Longman, Boston, MA; ISBN 0-201-74158-X; 2002; 759 pages.

7) Jones, Capers; Applied Software Measurement; 3rd edition; McGraw-Hill, New York, NY, 2008 (3rd edition due in 2008).

8) Jones, Capers; Estimating Software Costs; McGraw-Hill, New York, NY; 2007; (2nd edition.)

9) Jones, Capers; "Sizing Up Software"; Scientific American, New York, NY, December 1998; Vol. 279 No. 6; pp 104-109.

10) Jones, Capers; Software Assessments, Benchmarks, and Best Practices; Addison Wesley Longman, Boston, MA, 2000; 659 pages.

11) Jones, Capers; ; Conflict and Litigation Between Software Clients and Developers; Software Productivity Research, Burlington, MA; 2003; 54 pages.

12) Kan, Stephen H.; Metrics and Models in Software Quality Engineering, 2nd edition; Addison Wesley Longman, Boston, MA; ISBN 0-201-72915-6; 2003; 528 pages.

13) Putnam, Lawrence H.; Measures for Excellence -- Reliable Software On Time, Within Budget; Yourdon Press - Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-567694-0; 1992; 336 pages.

14) Putnam, Lawrence H and Myers, Ware.; Industrial Strength Software - Effective Management Using Measurement; IEEE Press, Los Alamitos, CA; ISBN 0-8186-7532-2; 1997; 320 pages.

15) Myers, Glenford et al; The Art of Software Testing; John Wiley & Sons, New York, NY; ISBN 0471469122; 2004.

Suggested Readings

Charette, Bob; Software Engineering Risk Analysis and Management; McGraw Hill, New York, NY; 1989.

Charette, Bob; Application Strategies for Risk Management; McGraw Hill, New York, NY; 1990.

DeMarco, Tom; Controlling Software Projects; Yourdon Press, New York; 1982; ISBN 0-917072-32-4; 284 pages.

Everett, Gerald D. and McLeod, Raymond; Software Testing – Testing Across the Entire Software Development Life Cycle; IEEE Press; 2007.

Ewusi-Mensah, Kweku; Software Development Failures; MIT Press, Cambridge, MA; 2003; ISBN 0-26205072-2276 pages.

Fernandini, Patricia L.; A Requirements Pattern; Addison Wesley, Boston, MA; 2002; ISBN 0-201-73826-0.

Flowers, Stephen; Software Failures: Management Failures; Amazing Stories and Cautionary Tales; John Wiley & Sons; 1996.

Galorath, Dan and Evans, Michael; Software Sizing, Estimation, and Risk Management: When Performance is Measured Performance Improves; Auerbach; Philadelphia, PA; 2006.

Garmus, David and Herron, David; Function Point Analysis – Measurement Practices for Successful Software Projects; Addison Wesley Longman, Boston, MA; 2001; ISBN 0-201-69944-3;363 pages.

Gibbs, T. Wayt; "Trends in Computing: Software's Chronic Crisis"; Scientific American Magazine, 271(3), International edition; pp 72-81; September 1994.

Gilb, Tom and Graham, Dorothy; Software Inspection; Addison Wesley, Harlow UK; 1993; ISBN 10: 0-201-63181-4.

Glass, R.L.; Software Runaways: Lessons Learned from Massive Software Project Failures; Prentice Hall, Englewood Cliffs; 1998.

International Function Point Users Group (IFPUG); IT Measurement – Practical Advice from the Experts; Addison Wesley Longman, Boston, MA; 2002; ISBN 0-201-74158-X; 759 pages.

Johnson, James et al; The Chaos Report; The Standish Group, West Yarmouth, MA; 2000.

Jones, Capers; Applied Software Measurement; McGraw Hill, 2nd edition 1996; ISBN 0-07-032826-9; 618 pages; 3rd edition due in the Spring of 2008.

Jones, Capers; Assessment and Control of Software Risks; Prentice Hall, 1994; ISBN 0-13-741406-4; 711 pages.

Jones, Capers; Patterns of Software System Failure and Success; International Thomson Computer Press, Boston, MA; December 1995; 250 pages; ISBN 1-850-32804-8; 292 pages.

Jones, Capers; Software Quality – Analysis and Guidelines for Success; International Thomson Computer Press, Boston, MA; ISBN 1-85032-876-6; 1997; 492 pages.

Jones, Capers; Estimating Software Costs; McGraw Hill, New York; 2007; ISBN 13-978-0-07-148300-1.

Jones, Capers; Software Assessments, Benchmarks, and Best Practices; Addison Wesley Longman, Boston, MA; ISBN 0-201-48542-7; 2000; 657 pages.

Jones, Capers: "Sizing Up Software;" Scientific American Magazine, Volume 279, No. 6, December 1998; pages 104-111.

Jones, Capers; Conflict and Litigation Between Software Clients and Developers; Software Productivity Research technical report; Narragansett, RI; 2007; 65 pages.

Kan, Stephen H.; Metrics and Models in Software Quality Engineering, 2nd edition; Addison Wesley Longman, Boston, MA; ISBN 0-201-72915-6; 2003; 528 pages.

Pressman, Roger; Software Engineering – A Practitioner's Approach; McGraw Hill, NY; 6th edition, 2005; ISBN 0-07-285318-2.

Radice, Ronald A.; High Qualitiy Low Cost Software Inspections; Paradoxicon Publishingl Andover, MA; ISBN 0-9645913-1-6; 2002; 479 pages.

Robertson, Suzanne and Robertson, James; Requirements-Led Project Management; Addison Wesley, Boston, MA; 2005; ISBN 0-321-18062-3.

Wiegers, Karl E.; Peer Reviews in Software – A Practical Guide; Addison Wesley Longman, Boston, MA; ISBN 0-201-73485-0; 2002; 232 pages.

Yourdon, Ed; Death March - The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects; Prentice Hall PTR, Upper Saddle River, NJ; ISBN 0-13-748310-4; 1997; 218 pages.

Yourdon, Ed; Outsource: Competing in the Global Productivity Race; Prentice Hall PTR, Upper Saddle River, NJ; ISBN 0-13-147571-1; 2005; 251 pages.

Web Sites

Information Technology Metrics and Productivity Institute (ITMPI): www.ITMPI.org

International Software Benchmarking Standards Group (ISBSG): www.ISBSG.org

International Function Point Users Group (IFPUG): www.IFPUG.org

Software Engineering Institute (SEI): www.SEI.org

Software Productivity Research (SPR): www.SPR.com