# Software Quality Metrics:

# Three Harmful Metrics and Two Helpful Metrics

June 6, 2012

Capers Jones, VP and Chief Technology Officer
Namcook Analytics LLC

Abstract

The cost of finding and fixing bugs or defects is the largest single expense element in the history of software. Bug repairs start with requirements and continue through development. After release, bug repairs and related customer support costs continue until the last user signs off. Over a 25 year life expectancy of a large software system in the 10,000 function point size range, almost 50 cents out of every dollar will go to finding and fixing bugs.

Given the fact that bug repairs are the most expensive element in the history of software, it might be expected that these costs would be measured carefully and accurately. They are not. Most companies do not measure defect repair costs, and when they do, they often use metrics that violate standard economic assumptions.

This article discusses three bad metrics and two good metrics. The three bad metrics are: 1) cost per defect; 2) lines of code; and 3) technical debt.

The two good metrics are: 1) function points, for normalization of data; and 2) Defect removal efficiency, for measuring the percentage of bugs found prior to release and afterwards via maintenance.

**Copyright © 2012 by Capers Jones. All rights reserved.**

## Introduction

The costs of finding and fixing bugs have been the major cost driver of large software applications since the software industry began. One might think that the software industry would have solid and reliable measurement data on its most expensive activity, but this is not the case.

Many companies do not start to measure bugs or defect repairs costs until testing starts, so all defects and repairs associated with requirements and design are invisible or under reported.

Even worse, many attempts to measure quality use metrics that violate standard economic assumptions, and conceal or distort the economic value of high quality. There are three very hazardous metrics that all distort quality economics and under state the true value of software quality:

1. Cost per defect;
2. Lines of code for normalization of results; and
3. Technical debt.

All three of these metrics share a common failing. They all ignore fixed costs, which will be dealt with later in this article. Individually each has other failings too.

There are two helpful and valid metrics that show the economic value of software quality and also can be used to predict quality and delivered defects, as well as measure:

1. Function points for normalization of results; and
2. Defect removal efficiency.

Let us start by considering the reasons that the three bad metrics are harmful, and then why the two good metrics are useful.

## The Errors and Hazards of Cost per Defect

The cost-per-defect metric has been in continuous use since the 1960's for examining the economic value of software quality. Hundreds of journal articles and scores of books include stock phrases, such as "*it costs 100 times as much to fix a defect after release as during early development.*"

Typical data for cost per defect varies from study to study but resembles the following pattern circa 2012:

| | |
|---|---|
| Defects found during requirements = | $250 |
| Defects found during design = | $500 |
| Defects found during coding and testing = | $1,250 |
| Defects found after release = | $5,000 |

While such claims are often true mathematically, there are five hidden problems with cost per defect that are usually not discussed in the software literature and are not well understood:

1. Cost per defect penalizes quality and is always cheapest where the greatest numbers of bugs are found.
2. Cost per defect ignores fixed costs. Even with zero defects there will be costs for inspections, testing, static analysis, and maintenance personnel. These costs are either fixed or inelastic and do not change at the same rate as defect volumes.

3. Because more bugs are found at the beginning of development than at the end, the increase in cost per defect is artificial. Actual time and motion studies of defect repairs show little variance from end to end, with some exceptions.
4. Even if calculated correctly, cost per defect does not measure the true economic value of improved software quality. Over and above the costs of finding and fixing bugs, high quality leads to shorter development schedules and overall reductions in development costs. These savings are not included in cost per defect calculations, so the metric understates the true value of quality by several hundred percent.
5. Cost per defect is not rigorous in distinguishing between valid unique defects and duplicate defect reports. High volume commercial packages often receive thousands of reports for the same bug. Even though the bug need only be fixed once, there are logistical costs for customer support, defect logging, and defect tracking that occur.

The cost per defect metric has such serious shortcomings for economic studies of software quality that it needs to be supplemented by additional information that distinguish between fixed and variable costs. Used by itself cost per defect penalizes quality and is cheapest for buggy applications and for bugs found early due to fixed costs.

Consider a well-known law of manufacturing economics:

*"If a manufacturing cycle includes a high proportion of fixed costs and there is a reduction in the number of units produced, the cost per unit will go up."*

As an application moves through a full test cycle that includes unit test, function test, regression test, performance test, system test, and acceptance test, the time required to write test cases and the time required to run test cases stays almost constant; but the number of defects found steadily decreases.

Table 1 shows the approximate costs for the three cost elements of preparation, execution, and repair for the test cycles just cited using a fixed rate $75.75 per hour for all activities.

- Writing test cases takes 16.5 hours for every test stage.
- Running test cases takes 9.9 hours for every test stage.
- Defect repair takes 5.0 hours for every defect found.

Even though every activity is based on fixed and unchanging amounts of time, look at what happens to cost per defect in Table 1:

**Table 1: Cost per Defect for Six Forms of Testing**
**(Assumes $75.75 per staff hour for costs)**

| | Writing Test Cases | Running Test Cases | Repairing Defects | TOTAL COSTS | Number of Defects | $ per Defect |
|---|---|---|---|---|---|---|
| Unit test | $1,250.00 | $750.00 | $18,937.50 | $20,937.50 | 50 | $418.75 |
| Function test | $1,250.00 | $750.00 | $7,575.00 | $9,575.00 | 20 | $478.75 |
| Regression test | $1,250.00 | $750.00 | $3,787.50 | $5,787.50 | 10 | $578.75 |
| Performance test | $1,250.00 | $750.00 | $1,893.75 | $3,893.75 | 5 | $778.75 |
| System test | $1,250.00 | $750.00 | $1,136.25 | $3,136.25 | 3 | $1,045.42 |
| Acceptance test | $1,250.00 | $750.00 | $378.75 | $2,378.75 | 1 | $2,378.75 |

What is most interesting about Table 1 is that cost per defect rises steadily as defect volumes come down, even though Table 1 uses a constant value of 5.0 hours to repair defects for every single test stage!

In other words every defect identified throughout Table 1 had a constant cost of $378.25 and 5 hours when only repairs are considered.

In fact, all three columns use constant values and the only true variable in the example is the number of defects found!

In real life, of course, preparation, execution, and repairs would all be variables. But by making them constant, it is easier to illustrate the main point: *cost per defect rises as numbers of defects decline*.

Since the main reason that cost per defect goes up as defects decline is due to the fixed costs associated with preparation and execution, it might be thought that those costs could be backed out and leave only defect repairs. Doing this would change the apparent results and minimize the initial errors, but it would introduce three new problems:

1. Removing quality cost elements that may total more than 50% of total quality costs would make it impossible to study quality economics with precision and accuracy.

2. Removing preparation and execution costs would make it impossible to calculate cost of quality (COQ) because the calculations for COQ demand all quality cost elements.

3. Removing preparation and execution costs would make it impossible to compare testing against formal inspections, because inspections do record preparation and execution as well as defect repairs.

Backing out or removing preparation and execution costs would be like going on a low-carb diet and not counting the carbs in pasta and bread, but only counting the carbs in meats and vegetables. The numbers might look good, but the results in real life would not be good.

The bottom line is that cost per defect penalizes quality and makes buggy applications look better than they are because their cost per defect is lower. Cost per defect also makes early defects look cheaper than late defects and has led to the urban legend that "*it costs 100 times as much to fix a bug after release than early in development.*"

Even worse, the true value of quality is not merely lowering defect repairs, but getting software out earlier, shortening development schedules, lowering maintenance costs, and having happier customers.

Cost per defect includes a hidden assumption that software will always contain many defects. In the future if software is constructed from collections of certified reusable components at the zero defect level there will still be costs for static analysis, critical feature inspection, and testing. Cost per defect does not envision or support zero-defect software.

In real life there are variations in defect discovery and repair costs. There are "abeyant defects" that sometimes take weeks to find and repair. What is needed is a more granular form of analysis that shows defect repairs by origin and the effort for finding and fixing bugs that originate in requirements, design, code, or other sources such as text materials and test cases.

Cost per defect has blinded the software industry to the true economic value of software and led to the false assumption that "high quality is expensive." High quality for software is not expensive, and in fact is much cheaper and faster to develop high quality software than buggy software.

## The Errors and Hazards of Lines of Code (LOC)

The "lines of code" or LOC metric has been in continuous use since the 1960's. Most users of LOC metrics have never studied the behavior of this metric across multiple languages.

As with the cost per defect metric, the LOC of code metric ignores fixed costs. The mathematical result is that low-level languages such as assembly and C seem to be cheaper and of higher quality than modern high-level languages such as Ruby and MySQL.

Let us consider two different languages to see what happens and why LOC metrics are so harmful to economic studies. We will consider the C language and the Java language as examples in table 2. We will assume that it takes twice as much C code as Java code for a specific application:

**Table 3: Quality Distortion caused by KLOC Metrics**

| | **C Language 2000 LOC (2 KLOC)** | **Java Language 1000 LOC (1 KLOC)** |
|---|---|---|
| **Defect Sources** | **Defects** | **Defects** |
| Requirements | 10 | 10 |
| Design | 20 | 20 |
| Source code | 30 | 15 |
| Documents | 3 | 3 |
| Bad fixes | 2 | 2 |
| TOTAL | 65 | 50 |
| Defects per KLOC | 32.50 | 50.00 |
| Defects Per Function Point | 3.25 | 2.50 |

Note that when data is normalized using "defects per KLOC" and all defect sources are included, the lower-level C language has fewer defects per KLOC. This is true even though the C version had 65 total defects and the Java version had only 50.

Even if only code defects are considered, there is still a distortion of results with LOC metrics. Code defects for both Java and C are exactly 15 per KLOC even though the C version had twice as many bugs.

LOC metrics have some legitimate uses, but they are not valid for software economic analysis and indeed should be considered to be professional malpractice for that purpose. LOC metrics can be used to examine coding speed, cyclomatic complexity, numbers of test cases, test coverage, and a number of ancillary topics. But LOC metrics are not suitable for economic studies.

The more languages that are included the worse LOC metric become. Following is Table 3.1 from an earlier study that compared 10 languages used for versions of a PBX switching system:

**Table 3.1: Productivity Rates for 10 Versions of the Same Software Project**
**(A PBX Switching system of 1,500 Function Points in Size)**

| Language | Effort (Months) | Funct. Pt. per Staff Month | Work Hrs. per Funct. Pt. | LOC per Staff Month | LOC per Staff Hour |
|---|---|---|---|---|---|
| Assembly | 781.91 | 1.92 | 68.81 | 480 | 3.38 |
| C | 460.69 | 3.26 | 40.54 | 414 | 3.13 |
| CHILL | 392.69 | 3.82 | 34.56 | 401 | 3.04 |
| PASCAL | 357.53 | 4.20 | 31.46 | 382 | 2.89 |
| PL/I | 329.91 | 4.55 | 29.03 | 364 | 2.76 |
| Ada83 | 304.13 | 4.93 | 26.76 | 350 | 2.65 |
| C++ | 293.91 | 5.10 | 25.86 | 281 | 2.13 |
| Ada95 | 269.81 | 5.56 | 23.74 | 272 | 2.06 |
| Objective C | 216.12 | 6.94 | 19.02 | 201 | 1.52 |
| Smalltalk | 194.64 | 7.71 | 17.13 | 162 | 1.23 |
| | | | | | |
| Average | 360.13 | 4.17 | 31.69 | 366 | 2.77 |

As can be seen, LOC metrics totally reverse real economic productivity and makes the most labor-intensive version using assembly language look faster than the most efficient version that used Smalltalk!

This is a textbook example of LOC as professional malpractice. This table comes from an actual consulting study where developers at a telecommunications company wanted to adopt object-oriented languages but management resisted because their internal LOC data made low-level languages look more productive than high-level languages!

It is a well-known law of manufacturing economics that when a manufacturing cycle has a high proportion of fixed costs and there is a reduction in the number of units manufactured, the cost per unit will go up.

If a "line of code" is selected as the manufacturing unit and there is a switch from a low-level language to a high-level language, the number of units will decrease. But the fixed costs of paperwork in requirements and specifications will not decrease. Therefore cost per line of code will always go up in direct proportion to the level of the language, with the very best languages looking the worst!

The costs of requirements, design, and other non-coding tasks on modern systems are often more expensive than the code itself. Of the five major cost drivers for software, LOC metrics can only be used for one. The five major cost elements are:

**Table 4: Major Software Cost Drivers 2012**

| | Activities | | % of Costs |
|---|---|---|---|
| 1 | Finding and fixing bugs | | 30.00% |
| 2 | Coding or programming | | 25.00% |
| 3 | Producing paper documents | | 20.00% |
| 4 | Meetings and communications | | 15.00% |
| 5 | Project management | | 10.00% |
| | | TOTAL | 100.00% |

LOC metrics have supplemental purposes for software projects, but should never be the primary metric for economic analysis.

**The Errors and Hazards of Technical Debt**

The concept of technical debt is the newest of the quality metrics, having first been described by Ward Cunningham in a 1992 paper. From that point on, the concept went viral and is now one of the most common quality metrics in the United States and indeed the world.

The essential idea of technical debt is that mistakes and errors made during development that escape into the real world when the software is released will accumulate downstream costs to rectify.

In a sense technical debt tends to piggyback on the "cost per defect" metric with an implied assumption that post-release defects and changes have higher costs than internal defects and changes.

As a metaphor or general concept the idea of technical debt is attractive and appealing. For one thing it makes software quality appear to take on some of the accumulated wisdom of financial operations, although the true financial understanding of the software industry is shockingly naive.

However technical debt suffers from the same problems as cost per defect and lines of code: it ignores fixed costs. It has other and much more serious problems that are not intrinsic, but have come to be unfortunately common.

A major problem with technical debt is that it ignores pre-release defect repairs, which are the major cost driver of almost all software applications. Ignoring pre-release defect repairs is a serious deficiency of technical debt.

Second, what happens after software is released to the outside world is not identical to the way software is developed. You need to support released software with customer support personnel who can handle questions and bug reports. And you also need to have maintenance programmers standing by to fix bugs when they are reported.

This means that even software with zero defects and very happy customers will accumulate post-release maintenance costs that are not accounted for by technical debt. Let us assume you release a commercial software application of 1,000 function points or 50,000 lines of Java code.

Prior to release you have trained 2 customer support personnel who are under contract and you have 1 maintenance programmer on your staff assigned to the new application. Thus even with zero defects you will have post-release costs of perhaps $15,000 per month.

After several months you can reassign the maintenance programmer and cut back to 1 customer support person, but the fact remains is that even zero-defect software has post-release costs.

The third and most serious flaw with technical debt concerns the 50% failure rate of large systems in the range of 10,000 function points or 500,000 Java statements in size. If an application of this size is cancelled and not released at all, then technical debt will of course be zero. But a company could lose $25,000,000 on a project that was terminated due to poor quality!

Yet another omission from the calculations for technical debt are the costs of litigation and punitive damages that might occur if disgruntled clients sue a vendor for poor quality.

Here is an example from an actual case. The shareholders of a major software company sued company management for releasing software of such poor quality that the shareholders claimed that poor quality was lowering the stock price.

Clearly the defects themselves would accumulate technical debt, but awards and punitive damages based on litigation are not included in technical debt calculations. In some cases, litigation costs, fines, and awards to the plaintiff might be high enough to bankrupt a software company.

This kind of situation is not included in the normal calculations for technical debt, but it should be. In other words, if technical debt is going to become a serious concept as is financial debt, then it needs to encompass every form of debt and not just post-release changes. It needs to encompass the high costs of cancelled projects and the even higher costs of losing major litigation for poor quality.

To illustrate that technical debt is only a partial measure of quality costs, Table 4 compares technical debt with cost of quality (COQ). As can be seen, technical debt only encompasses about 13% of the total costs of eliminating defects.

Note also that, while technical debt is shown as $86,141, right above this cost are the much higher costs of $428,625 for pre-release quality and defect repairs. These pre-release costs are not included in technical debt!

Just below technical debt are costs of $138,833 for fixed overhead costs of having support and maintenance people available. These overhead costs will accrue whether maintenance and support personnel are dealing with customer calls, fixing bugs, or just waiting for something to happen. Even with zero-defect software with zero technical debt there will still be overhead costs. These overhead costs are not included in technical debt, but are included in cost of quality (COQ).

**Table 4: Technical Debt Compared to Cost of Quality (COQ)**
**(1000 function points and 50,000 Java statements)**

| | | **Defects** | |
|---|---|---|---|
| **Code defect potential** | | **1,904** | |
| **Req. & design def. pot.** | | **1,869** | |
| **Total Defect Potential** | | **3,773** | |
| **Per function point** | | **3.77** | |
| **Per KLOC** | | **70.75** | |
| **Defect Prevention** | **Efficiency** | **Remainder** | **Costs** |
| JAD | 23% | **2,924** | $37,154 |
| QFD | 0% | **2,924** | $0 |
| Prototype | 20% | **2,340** | $14,941 |
| Models | 0% | **2,339** | $0 |
| Subtotal | **38%** | **2,339** | $52,095 |
| **Pre-Test Removal** | **Efficiency** | **Remainder** | **Costs** |
| Desk check | 25% | **1,755** | $19,764 |
| Static analysis | 55% | **790** | $20,391 |
| Inspections | 0% | **790** | $0 |
| Subtotal | **66%** | **790** | $40,155 |

| Test Removal | | Efficiency | Remainder | Costs |
|---|---|---|---|---|
| Unit | | 30% | **553** | $35,249 |
| Function | | 33% | **370** | $57,717 |
| Regression | | 12% | **326** | $52,794 |
| Component | | 30% | **228** | $65,744 |
| Performance | | 10% | **205** | $32,569 |
| System | | 34% | **135** | $69,523 |
| Acceptance | | 15% | **115** | $22,808 |
| | Subtotal | **85%** | **115** | $336,405 |

| | | Costs |
|---|---|---|
| **PRE-RELEASE COSTS** | | **$428,655** |
| **POST-RELEASE REPAIRS** | **(TECHNICAL DEBT)** | **$86,141** |
| **MAINTENANCE OVERHEAD** | | **$138,833** |
| **COST OF QUALITY (COQ)** | | **$653,629** |

| | |
|---|---|
| **Defects delivered** | **115** |
| **High severity** | **22** |
| **Security flaws** | **10** |
| **High severity %** | **18.94%** |

Even worse, if a software application is cancelled before release due to poor quality, it will have zero technical debt costs but a huge cost of quality.

An "average" project of 10,000 function points in size will cost about $20,000,000 to develop and about $5,000,000 to maintain for 5 years. About $3,000,000 of the maintenance costs will be technical debt.

But if a project of the same size is cancelled, they are usually late and over budget at the point of termination, so they might cost $26,000,000 that is totally wasted as a result of poor quality. Yet technical debt would be zero since the application was never released.

The bottom line is that the use of technical debt is an embarrassing revelation that the software industry does not understand basic economics. Cost of quality (COQ) is a better tool for quality economic study than technical debt.

## The Benefits of Function Point Metrics for Data Normalization

Function point metrics were developed Allan Albrecht and his colleagues at IBM and placed in the public domain in 1978. The International Function Point Users Group (IFPUG) took over the counting rules and function point training, and has grown into the largest measurement association in the world with affiliates in 24 countries.

In recent years a number of function point "clones" have been developed which differ slightly in counting rules. Among the many variants are COSMIC function points, FISMA function points, NESMA function points, function points light, engineering function points, feature points, and backfired function points.

There are also a number of specialized metrics that use some of the logic of function point analysis but mix in other counting rules. Two of the more common variants are use-case points and story points. Table 5 shows the comparative sizes of 15 functional metrics circa 2012:

**Table 5: Comparative Sizes of Functional Metrics Circa 2012**

| | Functional Metrics | Size | % of IFPUG |
|---|---|---|---|
| 1 | IFPUG function points | 1,000 | 100.00% |
| 2 | Backfired function points | 1,000 | 100.00% |
| 3 | Cosmic function points | 1,143 | 114.29% |
| 4 | Fast function points | 970 | 97.00% |
| 5 | Feature points | 1,000 | 100.00% |
| 6 | FISMA function points | 1,020 | 102.00% |
| 7 | Full function points | 1,170 | 117.00% |
| 8 | Function points light | 965 | 96.50% |
| 9 | Mark II function points | 1,060 | 106.00% |
| 10 | NESMA function points | 1,040 | 104.00% |
| 11 | RICE objects | 4,714 | 471.43% |
| 12 | SNAP non functional metrics | 235 | 23.53% |
| 13 | Story points | 556 | 55.56% |
| 14 | Unadjusted function points | 890 | 89.00% |
| 15 | Use case points | 333 | 33.33% |

In 2011, IFPUG issued new counting rules for non-functional size elements such as quality, performance, and the like. These are called "SNAP metrics" and are too new to have much empirical data.

The reason that IBM spent several million dollars inventing function points and the reason that function points are the most widely used metric in the world is that they actually demonstrate standard economic concepts. Function points can be used to normalize data in a fashion that matches standard economic assumptions.

Recall that Table 1 showed a significant increase in cost per defect throughout the testing cycle. This is the basis for the urban legend that it costs 100 times more to fix a bug after release than before.

Let us revisit the underlying data from Table 1 and see what happens when we normalize defect removal effort using cost per function point instead of cost per defect. Table 6 uses exactly the same effort used in Table 1:

- Writing test cases takes 16.5 hours for every test stage;
- Running test cases takes 9.9 hours for every test stage; and
- Defect repair takes 5.0 hours for every defect found.

Table 6 shows the results normalized using function points instead of cost per defect (but the underlying effort is identical in the two tables):

**Table 6 Cost per Function Point for Six Forms of Testing**
**(Assumes $75.75 per staff hour for costs)**

| | Writing Test Cases | Running Test Cases | Repairing Defects | TOTAL $ PER F.P. | Number of Defects |
|---|---|---|---|---|---|
| Unit test | $12.50 | $7.50 | $189.38 | $209.38 | 50 |
| Function test | $12.50 | $7.50 | $75.75 | $95.75 | 20 |

| | | | | | |
|---|---|---|---|---|---|
| Regression test | $12.50 | $7.50 | $37.88 | $57.88 | 10 |
| Performance test | $12.50 | $7.50 | $18.94 | $38.94 | 5 |
| System test | $12.50 | $7.50 | $11.36 | $31.36 | 3 |
| Acceptance test | $12.50 | $7.50 | $3.79 | $23.79 | 1 |

Notice the complete reversal of costs when function point metrics are used. Instead of becoming more expensive when few bugs found, defect removal costs per function point steadily decline as fewer and fewer bugs are found!

In other words defect repairs do not increase over time, they become cheaper over time. Table 6 is nothing more than the data from Table 1 with normalization based on cost per function point.

For the first time using function points, it is possible to actually study the economics of software quality in a fashion that matches standard economic assumptions, instead of distorting standard economic assumptions. This is why functional metrics are so powerful: they reveal real economic facts that are hidden by cost per defect, lines of code, and technical debt.

This article is based on IFPUG function points, but the same logic applies to COSMIC, FISMA, NESMA, and all the other function point variants. The only caveat is that the others will produce slightly different results.

The slow speed and high costs of manual function point counting have lowered the acceptance of these powerful metrics. As of 2012, several high-speed and low-cost function point methods are available that can reduce the costs of counting function points from more than $5.00 per function point counted down below $0.01 per function point counted. Within a few years these high-speed methods should make function points an industry standard.

**The Benefits of Defect Removal Efficiency (DRE)**

The most powerful and useful quality metric ever developed is that of "defect removal efficiency" (DRE). The reason for this claim is that improvements in DRE bring with them improvements in software schedules, software development costs, software maintenance costs, customer satisfaction, team morale, and stakeholder satisfaction. In other words, DRE is the central metric around which process improvements pivot.

DRE metrics were first developed inside IBM in the early 1970's as a method of evaluating the effectiveness of software inspections compared to software testing. As DRE usage expanded, it was found that DRE is perhaps the single most important software metric, because it is the best indicator of project health and also of development speed, costs, customer satisfaction, and quality.

Projects with DRE below 85% will always run late, will always be over budget, and will never have happy customers. On the other hand, projects with DRE above 95% will usually be on time, usually be under budget and usually have happy customers. No metric is perfect, but DRE is the best indicator of project health ever devised.

Defect removal efficiency is not too difficult to measure, nor is it an expensive metric. The essential math of DRE is to accumulate counts of all bugs during development. Then after 90 days of customer use, aggregate user defect reports with internal defect reports and calculate the percentage of bugs removed prior to release.

For example if your development team found 95 bugs before release and users reported 5 bugs in the first three months of usage, then DRE is obviously 95%.

(One caveat is that the International Software Benchmark Standards Group (ISBSG) uses only 30 days of customer usage in calculating DRE. Therefore, they always have higher DRE numbers than the author because 30 days of usage only reports about 20% of the bug volumes found in 90 days. Why ISBSG chose 30 days in unknown, since IBM and other companies have been using 90-day DRE measures since the early 1970's and the bulk of all published studies of DRE are based on 90 day windows.)

Table 7 illustrates two scenarios. Case A shows low quality without the use of pre-test inspections and static analysis. Case B shows high quality that includes the use of pre-test inspections and static analysis. Both Case A and Case B start with a defect potential of 1,000 defects. Case A uses only a standard sequence of testing. Case B uses pre-test static analysis and pre-test inspections before testing starts:

**Table 7: Examples of High and Low Defect Removal Efficiency**

| | **Case A** | | **Case B** | |
|---|---|---|---|---|
| | **Low Quality** | | **High Quality** | |
| **Defect Potential** | | 1,000 | | 1,000 |
| | **Efficiency** | | **Efficiency** | |
| **Pre-Test Removal** | | | | |
| Static analysis | 0.00% | 1,000 | 60.00% | 400 |
| Pre-Test inspection | 0.00% | 1,000 | 85.00% | 60 |
| **Test Removal** | | | | |
| Unit test | 25.00% | 750 | 30.00% | 42 |
| Function test | 27.00% | 548 | 33.00% | 28 |
| Regression test | 25.00% | 411 | 30.00% | 20 |
| Performance test | 12.00% | 361 | 17.00% | 16 |
| Component test | 33.00% | 242 | 37.00% | 10 |
| System test | 35.00% | 157 | 40.00% | 6 |
| Acceptance test | 15.00% | 134 | 15.00% | 5 |
| **Delivered defects** | | **134** | | **5** |
| **DRE** | **86.60%** | | | **99.50%** |

Note that pre-test inspections have a secondary benefit of raising testing efficiency levels. This is why testing efficiency is higher in Case B than in Case A.

Readers might think that while it is good to achieve high levels of defect removal efficiency, the costs might be prohibitive. This is a major economic misunderstanding by the software industry. High quality is not expensive. High quality is cheaper than low quality because testing costs are greatly reduced by pre-test inspections and static analysis. Table 8 show an approximate cost comparison of the differences between Case A and Case B:

**Table 8: Cost Comparison of Low Quality and High Quality**

| | Case A<br>Low Quality | Case B<br>High Quality |
|---|---:|---:|
| **Pre-Test Removal** | | |
| Static analysis | $0 | $5,000 |
| Pre-test inspection | $0 | $50,000 |
| | | |
| **Test Removal** | | |
| Unit test | $25,000 | $10,000 |
| Function test | $25,000 | $15,000 |
| Regression test | $10,000 | $5,000 |
| Performance test | $10,000 | $5,000 |
| Component test | $20,000 | $15,000 |
| System test | $25,000 | $10,000 |
| Acceptance test | $10,000 | $5,000 |
| | | |
| **Total Cost** | **$125,000** | **$120,000** |

In spite of the fact that pre-test inspections cost $50,000 the total cost of quality for the high-quality Case B is $5,000 less than the total cost of quality for the poor quality Case A.

The economic cost savings that accrue from high quality and high DRE cannot be measured using the three bad metrics of cost per defect, lines of code, and technical debt. But they can be measured using the combination of the two good metrics, function points and defect removal efficiency (DRE).

DRE is the most critical metric in all of software because it is the lynch pin of process improvements. Effective process improvement will raise DRE well above 95%. In fact a synergistic combination of pre-test static analysis, pre-test inspections, and formal mathematically-based testing can top 99% in DRE. Any methodology that does not measure and seek to improve DRE is essentially ineffective.

**Software Metrics Research Laboratories and Research Tools**

Ideally every proposed metric would be formally evaluated at a metrics research facility at a university or non-profit think tank. Unfortunately, software metrics just pop up like mushrooms after a rain without any formal evaluation or examination under controlled conditions.

To improve the rigor of metrics study, the author and Namcook Analytics LLC have built a metrics research tool. This tool, Software Risk Master™ (SRM) allows metrics to be evaluated under controlled conditions. For example SRM supports side-by-side analysis of cost per defect, function points, lines of code, technical debt, and DRE for the same application.

Users have the ability to specify exact quantities of defects in requirements, design, code, user documents, and bad fixes. Then they can follow both defect prevention and defect removal through any possible sequence of inspections, static analysis, and testing. Several of the tables in this report are taken from Software Risk Master™.

As defects are removed, costs are normalized using function points, cost per defect, lines of code in the KLOC format, and technical debt and the results are shown in a side-by-side format. This makes it easy to examine each metric in turn. Other metrics such as story points and use-case points can also be used.

The tool can also show the results of 32 different kinds of methodologies such as Agile, XP, pair-programming, RUP, TSP, EVO, Prince2, Merise, waterfall, modeling, iterative, spiral, etc.

Table 9 illustrates what some of the data looks like from a detailed Software Risk Master™ analysis. This only shows a few sample pre-test activities because a full analysis is too large for this article:

**Table 9: Defect Repair Costs by Origin and Activity**

| | **Require. Defects per Function Point** | **Design Defects per Function Point** | **Code Defects per Function Point** | **Document Defects per Function Point** | **TOTALS** |
|---|---|---|---|---|---|
| **Defect Potentials per FP** | **0.24** | **0.53** | **1.17** | **0.13** | **2.07** |
| **Defect potentials** | **181** | **399** | **874** | **96** | **1,550** |
| **Percent of total defects** | **11.67%** | **25.75%** | **56.39%** | **6.19%** | **100.00%** |
| **Security Vulnerabilities** | **0** | **0** | **1** | **0** | **1** |
| **PRE-TEST REMOVAL METHODS** | | | | | |
| | **Require. Efficiency** | **Design Efficiency** | **Code Efficiency** | **Document Efficiency** | **Total Efficiency** |
| **Text static analysis** | **50.00%** | **50.00%** | 0.00% | **50.00%** | 21.81% |
| Defects discovered | 90 | 200 | 0 | 48 | 338 |
| Bad-fix injection | 3 | 6 | 0 | 1 | 10 |
| **Defects remaining** | **93** | **206** | **874** | **49** | **1,222** |
| Team size | | | | | 2.50 |
| Schedule (months) | | | | | 0.16 |
| Defect logging, routing | $85 | $187 | $0 | $45 | $317 |
| Set up and running | $263 | $263 | $0 | $263 | $788 |
| Defect repairs | $707 | $1,559 | $0 | $375 | $2,641 |
| Repair integration/test | $57 | $125 | $0 | $30 | $211 |
| Text static cost | $1,110 | $2,134 | $0 | $713 | $3,957 |
| $ per function point | $1.48 | $2.85 | $0 | $0.95 | $5.28 |
| $ per KLOC | $27.76 | $53.34 | $0 | $17.82 | $98.92 |
| $ per defect | $12.28 | $10.69 | $0 | $14.84 | $11.70 |
| **Requirements inspection** | **87.00%** | 10.00% | 1.00% | 8.50% | 7.39% |
| Defects discovered | 81 | 21 | 9 | 4 | 115 |
| Bad-fix injection | 2 | 1 | 0 | 0 | 3 |
| **Defects remaining** | **15** | **186** | **866** | **45** | **1,111** |

| | | | | | |
|---|---|---|---|---|---|
| Team size | | | | | 4.88 |
| Schedule (months) | | | | | 0.97 |
| | | | | | |
| Defect logging, routing | $1,013 | $257 | $109 | $39 | $1,419 |
| Preparation/inspections | $8,203 | $7,031 | $5,859 | $3,516 | $24,609 |
| Defect repairs | $12,664 | $3,212 | $628 | $289 | $16,794 |
| Repair integration/test | $2,786 | $707 | $301 | $145 | $3,938 |
| Req. inspection cost | $24,667 | $11,207 | $6,897 | $3,989 | $46,760 |
| | | | | | |
| $ per function point | $32.89 | $14.94 | $9.20 | $5.32 | $62.35 |
| $ per KLOC | $616.67 | $280.18 | $172.44 | $99.71 | $1,169.01 |
| $ per defect | $304.33 | $545.12 | $789.00 | $948.78 | $408.18 |
| | | | | | |
| **Design inspection** | 40.00% | **87.00%** | 7.00% | 26.00% | 21.57% |
| Defects discovered | 6 | 162 | 61 | 12 | 240 |
| Bad-fix injection | 0 | 5 | 2 | 0 | 12 |
| **Defects remaining** | **9** | **29** | **807** | **34** | **879** |
| | | | | | |
| Team size | | | | | 4.88 |
| Schedule (months) | | | | | 1.33 |
| | | | | | |
| Defect logging, routing | $73 | $2,019 | $758 | $111 | $2,960 |
| Preparation/inspections | $7,031 | $8,203 | $4,688 | $2,344 | $22,266 |
| Defect repairs | $909 | $25,237 | $4,734 | $369 | $31,249 |
| Repair integration/test | $273 | $5,047 | $1,894 | $553 | $7,767 |
| Design inspection cost | $8,286 | $40,506 | $12,073 | $3,376 | $64,241 |
| | | | | | |
| $ per function point | $11.05 | $54.01 | $16.10 | $4.50 | $85.66 |
| $ per KLOC | $207.14 | $1,012.66 | $301.83 | $84.40 | $1,606.03 |
| $ per defect | $1,424.34 | $250.79 | $199.23 | $286.14 | $267.97 |

As can be seen costs are accumulated for defects from requirements, design, code, documents, and bad fixes. The details of defect removal costs are also very granular. A total of about 200 different forms of defect prevention, pre-test inspection, pre-test static analysis, and many kinds of testing can be evaluated in any combination and the results shown in absolute costs, and then normalized using cost per defect, cost per KLOC, and cost per function point.

Controlled results using the Software Risk Master™ tool show the hazards of the three bad metrics. The results indicate that cost per defect rise steadily and of course rises to infinity for zero-defect results. These results clearly violate standard economics.

The costs per line of code for defect removal are cheapest for assembly language and rise steadily when newer and more powerful languages are used such as Java, Ruby, Perl, Objective C, Smalltalk, and the like. Here too the results violate standard economics because defects are less numerous and repairs are cheaper with modern languages. In day to day usage LOC metrics also make requirements and design defects invisible.

For technical debt, the large costs of pre-release defect removal and the overhead costs of customer support and change teams are not included, so technical debt covers only a small percentage of cost of quality. For canceled projects that are not released due to poor quality, technical debt is zero in spite of huge losses.

With functional metrics, standard economics finally arrive in the software world and the true reduction in costs with better quality becomes visible. Functional metrics also work for pre-release defects and for overhead costs and even for canceled projects and litigation damages.

Simultaneous results are shown for all of these major metrics. Additional metrics such as use-case points and story points can also be tested in side-by-side form. But for the purposes of this article, SRM can be used to demonstrate the economic distortions of the three bad metrics using identical defect volumes and controlled sequences of defect removal activities.

## Summary and Conclusions about Software Quality Metrics

For more than 60 years, the software industry has lacked solid economic understanding of basic topics such as cost of quality and defect removal costs. The three bad metrics cited in this article distort economic reality and give the false impression that software quality is expensive, when in fact high quality is cheaper than poor quality.

In order to create valid economic models of software development, maintenance, and quality control it is urgent to have accurate measurements that use accurate metrics. The industry cannot afford the gaps and errors of bad metrics such as cost per defect, lines of code, and technical debt.

The combination of function point metrics combined with defect removal efficiency metrics (DRE) can show the true cost of quality and illustrate the fact that achieving high quality is the most cost-effective way to build software.

## References and Readings on Software Quality and Software Metrics

Boehm, Barry Dr.; Software Engineering Economics; Prentice Hall, Englewood Cliffs, NJ; 1981; 900 pages.

Booch Grady, Object Solutions: Managing the Object-Oriented Project; Addison Wesley, Reading, MA; 1995.

Bundschuh, Manfred and Dekkers, Carol; The IT Measurement Compendium; Springer; Berlin; 2008.

Capability Maturity Model Integration; Version 1.1; Software Engineering Institute; Carnegie-Mellon Univ.; Pittsburgh, PA; March 2003; http://www.sei.cmu.edu/cmmi/

Brooks, Fred: The Mythical Man-Month, Addison-Wesley, Reading, Mass., 1974, rev. 1995.

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.

Cohn, Mike; Agile Estimating and Planning; Prentice Hall PTR, Englewood Cliffs, NJ; 2005; ISBN 0131479415.

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

Ebert, Christof and Dumke, Reiner; Software Measurement: Establish, Extract, Evaluate, Execute; Springer, Berlin; 2007.

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

Gack, Gary; Managing the Black Hole – The Executives Guide to Project Risk; The Business Expert Publisher; Thomson, GA; 2010; ISBSG10: 1-935602-01-2.

Galorath, Dan; Software Sizing, Estimating, and Risk Management: When Performance is Measured Performance Improves; Auerbach Publishing, Philadelphia; 2006; ISBN 10: 0849335930; 576 pages.

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.

Gilb, Tom and Graham, Dorothy; Software Inspections; Addison Wesley, Reading, MA; 1993; ISBN 10: 0201631814.

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

Harris, Michael; Herron, David, and Iwaniciki, Stacia; The Business value of IT; Auerbach; 2008.

Hill, Peter R. Practical Software Project Estimation; McGraw Hill, 2010

Harris, Michael; Herron, David; and Iwanicki, Stacia; The Business Value of IT: Managing Risks, Optimizing Performance, and Measuring Results; CRC Press (Auerbach), Boca Raton, FL: ISBN 13: 978-1-4200-6474-2; 2008; 266 pages.

Humphrey, Watts; Managing the Software Process; Addison Wesley, Reading, MA; 1989.

Jones, Capers and Bonsignour, Olivier; *The Economics of Software Quality*; Addison Wesley, 2011; ISBN 978-0-13-258220-9; 57 pages.

Jones, Capers; Software Engineering Best Practices; McGraw Hill, 2009; ISBN 97800-07-162161-8; 660 pages.

Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008; ISBN 978-0-07-150244-3; 668 pages; 3rd edition (March 2008).

Jones, Capers; Estimating Software Costs; McGraw Hill, New York; 2nd edition, 2007; 644 pages; ISBN13: 978- 0-07-148300-1.

Jones, Capers; "*Estimating and Measuring Object-Oriented Software*"; American Programmer; 1994.

Jones, Capers; "*Why Flawed Software Projects are not Cancelled in Time*"; Cutter IT Journal; Vol. 10, No. 12; December 2003; pp. 12-17.

Jones, Capers; "*Software Project Management Practices: Failure Versus Success*"; Crosstalk, Vol. 19, No. 6; June 2006; pp4-8.

Jones, Capers; Software Assessments, Benchmarks, and Best Practices; Addison Wesley Longman, Boston, MA, 2000; 659 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; 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; Assessment and Control of Software Risks; Prentice Hall, 1994; ISBN 0-13-741406-4; 711 pages.

Jones, Capers; Conflict and Litigation Between Software Clients and Developers; Version 10; Software Productivity Research, Burlington, MA; June 2009; 54 pages.

Jones, Capers; A New Business Model for Function Points; Version 1.0; Capers Jones & Associates LLC; Narragansett, RI; June 2009; 40 pages.

Jones, Capers; A Short History of Lines-of-Code Metrics; Capers Jones & Associates LLC; Narragansett, RI; September 2009; 20 pages.

Jones, Capers; A Short History of Cost per Defect Metrics; Capers Jones & Associates LLC, Narragansett, RI; October 2009; 22 pages.

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

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.

McConnell; Software Estimating: Demystifying the Black Art; Microsoft Press, Redmund, WA; 2006.

Laird, Linda M and Brennan, Carol M; Software Measurement and Estimation: A Practical Approach; John Wiley & Sons, Hoboken, NJ; 2006; ISBN 0-471-67622-5; 255 pages.

Park, Robert E. et al; Software Cost and Schedule Estimating - A Process Improvement Initiative; Technical Report CMU/SEI 94-SR-03; Software Engineering Institute, Pittsburgh, PA; May 1994.

Park, Robert E. et al; Checklists and Criteria for Evaluating the Costs and Schedule Estimating Capabilities of Software Organizations; Technical Report CMU/SEI 95-SR-005; Software Engineering Institute, Pittsburgh, PA; January 1995.

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

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

Royce, Walker; Software Project Management: A Unified Framework; Addison Wesley, Reading, MA; 1998.

Roetzheim, William H. and Beasley, Reyna A.; Best Practices in Software Cost and Schedule Estimation; Prentice Hall PTR, Saddle River, NJ; 1998.

Strassmann, Paul; Information Productivity; Information Economics Press, Stamford, Ct; 1999.

Strassmann, Paul; Information Payoff; Information Economics Press, Stamford, Ct; 1985.

Strassmann, Paul; Governance of Information Management: The Concept of an Information Constitution; 2nd edition; (eBook); Information Economics Press, Stamford, Ct; 2004.

Strassmann, Paul; The Squandered Computer; Information Economics Press, Stamford, CT; 1997.

Stukes, Sherry, Deshoretz, Jason, Apgar, Henry and Macias, Ilona; Air Force Cost Analysis Agency Software Estimating Model Analysis ; TR-9545/008-2; Contract F04701-95-D-0003, Task 008; Management Consulting & Research, Inc.; Thousand Oaks, CA 91362; September 30 1996.

Symons, Charles R.: Software Sizing and Estimating—Mk II FPA (Function Point Analysis), John Wiley & Sons, Chichester, U.K., ISBN 0-471-92985-9, 1991.

Wellman, Frank, Software Costing: An Objective Approach to Estimating and Controlling the Cost of Computer Software, Prentice Hall, Englewood Cliffs, NJ, ISBN 0-138184364, 1992.

Whitehead, Richard; Leading a Development Team; Addison Wesley, Boston, MA; 2001; ISBN 10: 0201675267; 368 pages.

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

Various authors; The IFPUG Guide to IT and Software Measurement, Auerbach Publishers, April 2012; 828 pages.