Exceeding 99% in Defect Removal Efficiency (DRE) for Software

Draft 11.0 September 6, 2016

Capers Jones, VP and CTO, Namcook Analytics LLC

Abstract

Software quality depends upon two important variables. The first variable is that of "*defect potentials*" or the sum total of bugs likely to occur in requirements, architecture, design, code, documents, and "bad fixes" or new bugs in bug repairs. Defect potentials are measured using function point metrics, since "lines of code" cannot deal with requirements and design defects.

(This paper uses IFPUG function points version 4.3. The newer SNAP metrics are only shown experimentally due to insufficient empirical quality data with SNAP as of 2016. However an experimental tool is included for calculating SNAP defects.)

The second important measure is "*defect removal efficiency (DRE)*" or the percentage of bugs found and eliminated before release of software to clients.

The metrics of *Defect Potentials* and *Defect Removal Efficiency (DRE)* were developed by IBM circa 1973 and are widely used by technology companies and also by insurance companies, banks, and other companies with large software organizations.

The author's Software Risk Master (SRM) estimating tool predicts defect potentials and defect removal efficiency (DRE) as standard quality outputs for all software projects.

Web: www.Namcook.com
Email: Capers.Jones3@gmail.com

Introduction

Defect potentials and defect removal efficiency (DRE) are useful quality metrics developed by IBM circa 1973 and widely used by technology companies as well as by banks, insurance companies, and other organizations with large software staffs.

This combination of defect potentials using function points and defect removal efficiency (DRE) are the only accurate and effective measures for software quality. The "Cost per defect metric" penalizes quality and makes buggy software look better than high-quality software. The "Lines of code (LOC)" metric penalizes modern high-level languages. The LOC metric can't measure or predict bugs in requirements and design. The new technical debt metric only covers about 17% of the true costs of poor quality.

Knowledge of effective software quality control has major economic importance because for over 50 years the #1 cost driver for the software industry has been the costs of finding and fixing bugs. Table 1 shows the 15 major cost drivers for software projects in 2016. The cost drivers highlighted in red are attributable to poor software quality:

Table 1: U.S. Software Costs in Rank Order:

1) The cost of finding and fixing bugs
2) The cost of cancelled projects
3) The cost of producing English words
4) The cost of programming or code development
5) The cost of requirements changes
6) The cost of successful cyber-attacks
7) The cost of customer support
8) The cost of meetings and communication
9) The cost of project management
10) The cost of renovation and migration
11) The cost of innovation and new kinds of software
12) The cost of litigation for failures and disasters
13) The cost of training and learning
14) The cost of avoiding security flaws
15) The cost of assembling reusable components

Table 1 illustrates an important but poorly understood economic fact about the software industry. Four of the 15 major cost drivers can be attributed specifically to poor quality. The poor quality of software is a professional embarrassment and a major drag on the economy of the software industry and for that matter a drag on the entire U.S. and global economies.

Poor quality is also a key reason for cost driver #2. A common reason for cancelled software projects is because quality is so bad that schedule slippage and cost overruns turned the project return on investment (ROI) from positive to negative.

Note the alarming location of successful cyber-attacks in 6th place (and rising) on the cost-drive list. Since security flaws are another form of poor quality it is obvious that high quality is needed to deter successful cyber-attacks.

Poor quality is also a key factor in cost driver #12 or litigation for breach of contract. (The author has worked as an expert witness in 15 lawsuits. Poor software quality is an endemic problem with breach of contract litigation. In one case against a major ERP company, the litigation was filed by the company's own shareholders who asserted that the ERP package quality was so bad that it was lowering stock values!)

A chronic weakness of the software industry for over 50 years has been poor measurement practices and bad metrics for both quality and productivity. For example many companies don't even start quality measures until late testing, so early bugs found by inspections, static analysis, desk checking, and unit testing are unmeasured and invisible.

If you can't measure a problem then you can't fix the problem either. Software quality has been essentially unmeasured and therefore unfixed for 50 years. This paper shows how quality can be measured with high precision, and also how quality levels can be improved by raising defect removal efficiency (DRE) up above 99%, which is where it should be for all critical software projects.

Software defect potentials are the sum total of bugs found in requirements, architecture, design, code, and other sources of error. The approximate U.S. average for defect potentials is shown in table 2 using IFPUG function points version 4.3:.

Table 2: Average Software Defect Potentials circa 2016 for the United States

- Requirements 0.70 defects per function point
- Architecture 0.10 defects per function point
- Design 0.95 defects per function point
- Code 1.15 defects per function point
- Security code flaws 0.25 defects per function point
- Documents 0.45 defects per function point
- Bad fixes 0.65 defects per function point
- **Totals** **4.25 defects per function point**

Note that the phrase "bad fix" refers to new bugs accidentally introduced in bug repairs for older bugs. The current U.S. average for bad-fix injections is about 7%; i.e. 7% of all bug repairs contain new bugs. For modules that are high in cyclomatic complexity and for "error prone modules" bad fix injections can top 75%. For applications with low cyclomatic complexity bad fixes can drop below 0.5%.

Defect potentials are of necessity measured using function point metrics. The older "lines of code" metric cannot show requirements, architecture, and design defects not any other defect outside the code itself. (As of 2016 function points are the most widely used software metric in the world. There are more benchmarks using function point metrics than all other metrics put together.)

Because of the effectiveness of function point measures compared to older LOC measures an increasing number of national governments are now mandating function point metrics for all software contracts. The governments of Brazil, Italy, Japan, Malaysia and South Korea now require function points for government software. Table 3 shows the countries with rapid expansions in function point use:

Table 3 Countries Expanding Use of Function Points 2016

1	Argentina	
2	Australia	
3	Belgium	
4	**Brazil**	**Required for government contracts**
5	Canada	
6	China	
7	Finland	
8	France	
9	Germany	
10	India	
11	**Italy**	**Required for government contracts**
12	**Japan**	**Required for government contracts**
13	**Malaysia**	**Required for government contracts**
14	Mexico	
15	Norway	
16	Peru	
17	Poland	
18	Singapore	
19	**South Korea**	**Required for government contracts**
20	Spain	
21	Switzerland	
22	Taiwan	
23	The Netherlands	
24	United Kingdom	
25	United States	

To be blunt, any company or government agency in the world that does not use function point metrics does not have accurate benchmark data on either quality or productivity. The software industry has had poor quality for over 50 years and a key reason for this problem is that the

software industry has not measured quality well enough make effective improvements. Cost per defect and lines of code both distort reality and conceal progress. They are harmful rather than helpful in improving either quality or productivity.

Lines of code reverses true economic productivity and makes assembly language seem more productive than Objective C. Cost per defect reverses true quality economics and makes buggy software look cheaper than high quality software. These distortions of economic reality have slowed software progress for over 50 years.

The U.S. industries that tend to use function point metrics and therefore understand software economics fairly well include automotive manufacturing, banks, commercial software, insurance, telecommunications, and some public utilities.

For example Bank of Montreal was one of the world's first users of function points after IBM placed the metric in the public domain; Ford has used function point metrics for fuel injection and navigation packages; Motorola has used function points for smart phone applications; AT&T has used function points for switching software; IBM has used function points for both commercial software and also operating systems.

The U.S. industries that do not use function points widely and hence have no accurate data on either software quality or productivity include the Department of Defense, most state governments, the U.S. Federal government, and most universities (which should understand software economics but don't seem to.)

Although the Department of Defense was proactive in endorsing the Software Engineering Institute (SEI) capability maturity model integrated (CMMI), it lags the civilian sector in software metrics and measurements. For that matter the SEI itself has not yet supported function point metrics nor pointed out to clients that both lines of code and cost per defect distort reality and reverse the true economic value of high quality and high-level programming languages.

It is interesting that the author had a contract from the U.S. Air Force to examine the benefits of ascending to the higher CMMI levels because the SEI itself had no quantitative data available. In fact the findings from this study are shown later in this report in Table 12.

Although the Department of Defense itself lags in function point use some of the military services have used function points for important projects. For example the U.S. Navy has used function points for shipboard gun controls and cruise missile navigation.

If a company or government agency wants to get serous in improving quality then the best and only effective metrics for achieving this are the combination of defect potentials in function points and defect removal efficiency (DRE).

Defect removal efficiency (DRE) is calculated by keeping accurate counts of all defects found during development. After release all customer-reported bugs are included in the total. After 90

days of customer usage DRE is calculated. If developers found 900 bugs and customer reported 50 bugs in the first three months then DRE is 95%.

Obviously bug reports don't stop cold after 90 days, but the fixed 90-day interval provides an excellent basis for statistical quality reports.

The overall range in defect potentials runs from about 2.00 per function point to more than 7.00 per function point. Factors that influence defect potentials include team skills, development methodologies, CMMI levels, programming languages, and defect prevention techniques such as joint application design (JAD) and quality function deployment (QFD).

Some methodologies such as team software process (TSP) are "quality strong" and have low defect potentials.) Agile is average for defect potentials. Waterfall is worse than average for defect potentials.

Table 4 shows the U.S. ranges for defect potentials circa 2016:

Table 4: U.S Average Ranges of Defect Potentials Circa 2016 (Defects per IFPUG 4.3 function point)

Defect Origins	**Best**	**Average**	**Worst**
Requirements	0.34	0.70	1.35
Architecture	0.04	0.10	0.20
Design	0.63	0.95	1.58
Code	0.44	1.15	2.63
Security flaws	0.18	0.25	0.40
Documents	0.20	0.45	0.54
Bad fixes	0.39	0.65	1.26
TOTAL	**2.21**	**4.25**	**7.95**

NOTE: the author's Software Risk Master (SRM) estimating tool predicts defect potentials as a standard output for every project estimated.

Defect potentials obviously vary by size, with small projects typically having low defect potentials. Defect potentials rise faster than size increases, with large systems above 10,000 function points having alarmingly high defect potentials.

Table 5 shows U.S. ranges in defect potentials from small projects of 1 function point up to massive systems of 100,000 function points:

Table 5: Software Defect Potentials per Function Point by Size (Defects per IFPUG 4.3 function point)

Function Points	Best	Average	Worst
1	0.60	1.50	2.55
10	1.25	2.50	4.25
100	1.75	3.25	6.13
1000	2.14	4.75	8.55
10000	3.38	6.50	12.03
100000	4.13	8.25	14.19
Average	**2.21**	**4.25**	**7.95**

As can be seen defect potentials go up rapidly with application size. This is one of the key reasons why large systems fail so often and also run late and over budget.

Table 6 shows the overall U.S. ranges in defect removal efficiency (DRE) by applications size from a size of 1 function point up to 100,000 function points. As can be seen DRE goes down as size goes up:

Table 6: U.S. Software Average DRE Ranges by Application Size

Function Points	Best	Average	Worst
1	99.90%	97.00%	94.00%
10	99.00%	96.50%	92.50%
100	98.50%	95.00%	90.00%
1000	96.50%	94.50%	87.00%
10000	94.00%	89.50%	83.50%
100000	91.00%	86.00%	78.00%
Average	**95.80%**	**92.20%**	**86.20%**

Table 7 is a somewhat complicated table that combines the results of tables 5 and 6; i.e. both defect potentials and defect removal efficiency (DRE) ranges are now shown together on the same table. Note that as size increases defect potentials also increase, but defect removal efficiency (DRE) comes down:

Table 7: Software Defect Potentials and DRE Ranges by Size

Function Points		Best	Average	Worst
1	Defect Potential	**0.60**	1.50	**2.55**
	DRE	**99.90%**	97.00%	**94.00%**
	Delivered defects	**0.00**	0.05	**0.15**
10	Defect Potential	**1.25**	2.50	**4.25**
	DRE	**99.00%**	96.00%	**92.50%**
	Delivered defects	**0.01**	0.10	**0.32**
100	Defect Potential	**1.75**	3.50	**6.13**
	DRE	**98.50%**	95.00%	**90.00%**
	Delivered defects	**0.03**	0.18	**0.61**
1000	Defect Potential	**2.14**	4.75	**8.55**
	DRE	**96.50%**	94.50%	**87.00%**
	Delivered defects	**0.07**	0.26	**1.11**
10000	Defect Potential	**3.38**	6.50	**12.03**
	DRE	**94.00%**	89.50%	**83.50%**
	Delivered defects	**0.20**	0.68	**1.98**
100000	Defect Potential	**4.13**	8.25	**14.19**
	DRE	**91.00%**	86.00%	**78.00%**
	Delivered defects	**0.37**	1.16	**3.12**

Best-case results are usually found for software controlling medical devices or complex physical equipment such as aircraft navigation packages, weapons systems, operating systems, or telecommunication switching systems. These applications are usually large and range from about 1000 to over 100,000 function points in size. Large complex applications require very high DRE levels in order for the physical equipment to operate safely. They normally use pre-test inspections and static analysis and usually at least 10 test stages.

Average-case results are usually found among banks, insurance companies, manufacturing, and commercial software. These applications are also on the large size and range from 1000 to more than 10,000 function points. Here too high levels of DRE are important since these applications contain and deal with confidential data. These applications normally use pre-test static analysis and at least 8 test stages.

Worst-case results tend to show up in litigation for cancelled projects or for lawsuits for poor quality. State, municipal, and civilian Federal government software projects, and especially large systems such a taxation, child support, and motor vehicles are often in the worst-case class.

It is an interesting point that every lawsuit where the author has worked as an expert witness has been for large systems > 10,000 function points in size. These applications seldom use either pre-test inspections or static analysis and sometimes use only 6 test stages.

While function point metrics are the best choice for normalization, it is also important to know the actual numbers of defects that are likely to be present when software applications are delivered to customers. Table 8 shows data from table 7 only expanded to show total numbers of delivered defects:

Table 8: U.S. Average Delivered Defects by Application Size

Function Points	Best	Average	Worst
1	0	0	1
10	0	1	3
100	3	18	61
1000	75	261	1,112
10000	2,028	6,825	19,841
100000	3,713	11,550	31,218
Average	**970**	**3,109**	**8,706**

Here too it is painfully obvious that defect volumes go up with application size. However table 8 shows all severity levels of delivered defects. Only about 1% of delivered defects will be in the high-severity class of 1 and only about 14% in severity class 2. Severity class 3 usually has about 55% while severity 4 has about 30%.

Defect potentials have also varied by decade. Table 9 shows approximate values starting in 1960 and ending with projected values for 2019. The reason for the gradual improvement in defect potentials include the advent of newer programming languages, the average increase in organizations with higher CMMI levels, a gradual decrease in application size, and a gradual increase on reusable materials from older applications.

Table 9: Defect Potentials by Decade

	Best	Average	Worst
1960-1969	2.85	5.50	10.29
1970-1979	2.72	5.25	9.82
1980-1989	2.59	5.00	9.35
1990-1999	2.46	4.75	8.88
2000-2009	2.33	4.50	8.42
2010-2019	2.20	4.25	7.95

These severity levels are normally assigned by software quality assurance personnel. Because companies fix high severity bugs faster than low severity bugs, clients often report bugs as being severity 2 that are really only severity 3 or severity 4. While the IBM average for severity 2 bugs was about 14%, clients tend to exaggerate and rank over 50% of bug reports as severity 2!

This classification of defect severity levels was developed by IBM circa 1960: It has been used for over 50 years by thousands of companies for hundreds of thousands of software applications.

Table 10: IBM Defect Severity Scale (1960 – 2016)

Severity 1 Software does not work at all

Severity 2 Major features disabled and inoperative

Severity 3 Minor bug that does not prevent normal use

Severity 4 Cosmetic errors that do not affect operation

Invalid Defects not correctly reported; i.e. hardware problems reported as software

Duplicate Multiple reports of the same bug

Abeyant Unique defects found by only 1 client that cannot be duplicated

It is obvious that valid high-severity defects of severities 1 and 2 are the most troublesome for software projects.

Defect removal efficiency (DRE) is a powerful and useful metric. Every important project should measure DRE and every important project should top 99% in DRE, but few do.

As defined by IBM circa 1973 DRE is measured by keeping track of all bugs found internally during development, and comparing these to customer-reported bugs during the first 90 days of usage. If internal bugs found during development total 95 and customers report 5 bugs in the first three months of use then DRE is 95%.

Another important quality topic is that of "error-prone modules" (EPM) also discovered by IBM circa 1970. IBM did a frequency analysis of defect distributions and was surprised to find that bugs are not randomly distributed, but clump in a small number of modules. For example in the IBM IMS data base application there were 425 modules. About 300 of these were zero-defect modules with no customer-reported bugs. About 57% of all customer reported bugs were noted in only 31 modules out of 425. These tended to be high in cyclomatic complexity, and also had failed to use pre-test inspections. Table 11 shows approximate results for EPM in software by application size:

Table 11: Distribution of "Error Prone Modules" (EPM) in Software

Function Points	Best	Average	Worst
1	0	0	0
10	0	0	0
100	0	0	0
1000	0	2	4
10000	0	18	49
100000	0	20	120
Average	**0**	**7**	**29**

EPM were discovered by IBM but unequal distribution of bugs was also noted by many other companies whose defect tracking tools can highlight bug reports by modules. For example EPM were confirmed by AT&T, ITT, Motorola, Boeing, Raytheon, and other technology companies with detailed defect tracking systems.

EPM tend to resist testing, but are fairly easy to find using pre-test static analysis, pre-test inspections, or both. EPM are treatable, avoidable conditions and should not be allowed to occur in modern software circa 2016. The presence of EPM is a sign of inadequate defect quality measurements and inadequate pre-test defect removal activities.

The author had a contract from the U.S. Air Force to examine the value of ascending to the higher levels of the capability maturity model integrated (CMMI). Table 12 shows the approximate quality results for all five levels of the CMMI:

Table 12: Software Quality and the SEI Capability Maturity Model Integrated (CMMI) for 2,500 function points

CMMI Level	Defect Potential per Function Point	Defect Removal Efficiency	Delivered Defects per Function Point	Delivered Defects
SEI CMMI 1	4.50	87.00%	0.585	1,463
SEI CMMI 2	3.85	90.00%	0.385	963
SEI CMMI 3	3.00	96.00%	0.120	300
SEI CMMI 4	2.50	97.50%	0.063	156
SEI CMMI 5	2.25	99.00%	0.023	56

Table 12 was based on study by the author commissioned by the U.S. Air Force. Usage of the CMMI is essentially limited to military and defense software. Few civilian companies use the CMMI and the author has met several CIO's from large companies and state governments that have never even heard of SEI or the CMMI.

Software defect potentials and DRE also vary by industry. Table 13 shows a sample of 15 industries with higher than average quality levels out of a total of 75 industries where the author has data:

Table 13: Software Quality Results by Industry

	Industry	Defect Potentials per Function Point 2016	Defect Removal Efficiency 2016	Delivered Defects per Function Pt 2016
	Best Quality			
1	Manufacturing - medical devices	4.60	99.50%	0.02
2	Manufacturing - aircraft	4.70	99.00%	0.05
3	Government - military	4.70	99.00%	0.05
4	Smartphone/tablet applications	3.30	98.50%	0.05
5	Government - intelligence	4.90	98.50%	0.07
6	Software (commercial)	3.50	97.50%	0.09
7	Telecommunications operations	4.35	97.50%	0.11
8	Manufacturing - defense	4.65	97.50%	0.12
9	Manufacturing - telecommunications	4.80	97.50%	0.12
10	Process control and embedded	4.90	97.50%	0.12
11	Manufacturing - pharmaceuticals	4.55	97.00%	0.14
12	Professional support - medicine	4.80	97.00%	0.14
13	Transportation - airlines	5.87	97.50%	0.15
14	Manufacturing - electronics	4.90	97.00%	0.15
15	Banks - commercial	4.15	96.25%	0.16

There are also significant differences by country. Table 14 shows a sample of 15 countries with better than average quality out of a total of 70 countries where the author has data:

Table 14: Samples of Software Quality by Country

	Countries	Defect Potential per FP 2016	Defect Removal Efficiency (DRE) 2016	Delivered Defects per Function Pt 2016
	Best Quality			
1	Japan	4.25	96.00%	0.17
2	India	4.90	95.50%	0.22
3	Finland	4.40	94.50%	0.24
4	Switzerland	4.40	94.50%	0.24
5	Denmark	4.25	94.00%	0.26
6	Israel	5.00	94.80%	0.26
7	Sweden	4.45	94.00%	0.27
8	Netherlands	4.40	93.50%	0.29
9	Hong Kong	4.45	93.50%	0.29
10	Brazil	4.50	93.00%	0.32
11	Singapore	4.80	93.40%	0.32

12	United Kingdom	4.55	93.00%	0.32
13	Malaysia	4.60	93.00%	0.32
14	Norway	4.65	93.00%	0.33
15	Taiwan	4.90	93.30%	0.33

Countries such as Japan and India tend to be more effective in pre-test defect removal operations and to use more certified test personnel than those lower down the table. Although not shown in table 14 the U.S. ranks as country #19 out of the 70 countries from which the author has data.

Table 15 shows quality comparison of 15 software development methodologies (this table is cut down from a larger table of 80 methodologies that will be published in the author's next book.)

Table15: Comparisons of 15 Software Methodologies

	Methodologies	Defect Potential per FP 2016	Defect Removal Efficiency 2016	Delivered Defects per FP 2016
	Best Quality			
1	Reuse-oriented (85% reusable materials)	1.30	99.50%	0.007
2	Pattern-based development	1.80	99.50%	0.009
3	Animated, 3D, full color design development	1.98	99.20%	0.016
4	Team software process (TSP) + PSP	2.35	98.50%	0.035
5	Container development (65% reuse)	2.90	98.50%	0.044
6	Microservice development	2.50	98.00%	0.050
7	Model-driven development	2.60	98.00%	0.052
8	Microsoft SharePoint development	2.70	97.00%	0.081
9	Mashup development	2.20	96.00%	0.088
10	Product Line engineering	2.50	96.00%	0.100
11	DevOps development	3.00	94.00%	0.180
12	Pair programming development	3.10	94.00%	0.186
13	Agile + scrum	3.20	92.50%	0.240
14	Open-source development	3.35	92.00%	0.268
15	Waterfall development	4.60	87.00%	0.598

Table 16 shows the details of how defect removal efficiency (DRE) operates. Table 16 must of course use fixed values but there are ranges for every row and column for both pre-test and test methods.

There are also variations in the numbers of pre-test removal and test stages used. Table 16 illustrates the maximum number observed.

The data in table 16 is originally derived from IBM"s software quality data collection which is more complete than most companies. Other companies have been studied as well. Note that requirements defects are among the most difficult to remove since they are resistant to testing.

To consistently top 99% in DRE the minimum set of methods needed include most of the following:

Pre-Test Removal

1. Formal Inspections (requirements, design, code, etc.)
2. Code Static analysis
3. Automated Requirements modeling
4. Automated correctness proofs

Test Removal

1. Unit test (manual/automated)
2. Function test
3. Regression test
4. Integration test
5. Performance test
6. Usability test
7. Security test
8. System test
9. Field or acceptance test

In other words a series of about 13 kinds of defect removal activities are generally needed to top 99% in DRE consistently. Testing by itself without inspections or static analysis usually is below 90% in DRE.

Of course some critical applications such as medical devices and weapons systems use many more kinds of testing. As many as 18 kinds of testing have been observed by the author. This paper uses 12 kinds of testing since these are fairly common on large systems > 10,000 function points in size which is where quality is a critical factor.

Note that DRE includes bugs that originate in architecture, requirements, design, code, documents, and "bad fixes" or new bugs in bug repairs themselves. All bug origins should be included since requirements and design bugs often outnumber code bugs.

Note that the defect potential for next table 16 is somewhat lower than the 4.25 value shown in tables 1, 2, and 3. This is because those tables includes all programming languages and some have higher defect potentials than Java, which is used for table 16.

Code defect potentials vary by language with low-level languages such as assembly and C having a higher defect potential than high-level languages such as Java, Objective C, C#, Ruby, Python, etc.

Table16: Software Quality and Defect Removal Efficiency (DRE)
Note 1: The table represents high quality defect removal operations.

Application size in function points	**1,000**
Application language	Java
Source lines per FP	53.33
Source lines of code	53,330

	Pre-Test Defect Removal Methods	Architect. Defects per Function Point	Require. Defects per Function Point	Design Defects per Function Point	Code Defects per Function Point	Document Defects per Function Point	TOTALS
	Defect Potentials per Function Point	**0.25**	**1.00**	**1.15**	**1.30**	**0.45**	**4.15**
	Defect potentials	**250**	**1,000**	**1,150**	**1,300**	**450**	**4,150**
1	**Requirement inspection**	5.00%	**87.00%**	10.00%	5.00%	8.50%	26.52%
	Defects discovered	13	870	115	65	38	1,101
	Bad-fix injection	0	26	3	2	1	33
	Defects remaining	237	104	1,032	1,233	411	3,016
2	**Architecture inspection**	**85.00%**	10.00%	10.00%	2.50%	12.00%	13.10%
	Defects discovered	202	10	103	31	49	395
	Bad-fix injection	6	0	3	1	1	12
	Defects remaining	30	93	925	1,201	360	2,609
3	**Design inspection**	10.00%	14.00%	**87.00%**	7.00%	16.00%	36.90%
	Defects discovered	3	13	805	84	58	963
	Bad-fix injection	0	0	24	3	2	48
	Defects remaining	26	80	96	1,115	301	1,618
4	**Code inspection**	12.50%	15.00%	20.00%	**85.00%**	10.00%	62.56%
	Defects discovered	3	12	19	947	30	1,012
	Bad-fix injection	0	0	1	28	1	30
	Defects remaining	23	67	76	139	270	575
5	**Code Static Analysis**	2.00%	2.00%	7.00%	**55.00%**	3.00%	15.92%
	Defects discovered	0	1	5	76	8	92
	Bad-fix injection	0	0	0	2	0	3
	Defects remaining	23	66	71	60	261	481
6	**IV & V**	10.00%	12.00%	**23.00%**	7.00%	18.00%	16.16%

	Defects discovered	2	8	16	4	47	78
	Bad-fix injection	0	0	0	0	1	2
	Defects remaining	20	58	54	56	213	401
7	**SQA review**	10.00%	17.00%	17.00%	12.00%	12.50%	30.06%
	Defects discovered	2	10	9	7	27	54
	Bad-fix injection	0	0	0	0	1	3
	Defects remaining	18	48	45	49	185	344
	Pre-test defects removed	**232**	**952**	**1,105**	**1,251**	**265**	**3,805**
	Pre-test efficiency %	**92.73%**	**95.23%**	**96.12%**	**96.24%**	**58.79%**	**91.69%**

Test Defect Removal Stages

		Architect.	**Require.**	**Design**	**Code**	**Document**	**Total**
1	**Unit testing (Manual)**	2.50%	4.00%	7.00%	**35.00%**	10.00%	11.97%
	Defects discovered	0	2	3	17	19	41
	Bad-fix injection	0	0	0	1	1	1
	Defects remaining	18	46	41	31	166	301
2	**Function testing**	7.50%	5.00%	22.00%	**37.50%**	10.00%	13.63%
	Defects discovered	1	2	9	12	17	41
	Bad-fix injection	0	0	0	0	0	1
	Defects remaining	16	43	32	19	149	259
3	**Regression testing**	2.00%	2.00%	5.00%	**33.00%**	7.50%	7.84%
	Defects discovered	0	1	2	6	11	20
	Bad-fix injection	0	0	0	0	0	1
	Defects remaining	16	43	30	13	138	238
4	**Integration testing**	6.00%	20.00%	22.00%	**33.00%**	15.00%	17.21%
	Defects discovered	1	9	7	4	21	41
	Bad-fix injection	0	0	0	0	1	1
	Defects remaining	15	34	23	8	116	196
5	**Performance testing**	14.00%	2.00%	**20.00%**	18.00%	2.50%	6.07%
	Defects discovered	2	1	5	2	3	12
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	13	33	19	7	113	184
6	**Security testing**	12.00%	15.00%	**23.00%**	8.00%	2.50%	7.71%
	Defects discovered	2	5	4	1	3	14
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	11	28	14	6	110	169
7	**Usability testing**	12.00%	17.00%	15.00%	5.00%	**48.00%**	36.42%
	Defects discovered	1	5	2	0	53	62
	Bad-fix injection	0	0	0	0	2	2
	Defects remaining	10	23	12	6	56	106

8	**System testing**	16.00%	12.00%	18.00%	12.00%	**34.00%**	24.81%
	Defects discovered	2	3	2	1	19	26
	Bad-fix injection	0	0	0	0	1	1
	Defects remaining	8	20	10	5	36	79
9	**Cloud testing**	10.00%	5.00%	13.00%	10.00%	**20.00%**	13.84%
	Defects discovered	1	1	1	1	7	11
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	7	19	8	5	29	69
10	**Independent testing**	12.00%	10.00%	11.00%	10.00%	**23.00%**	15.81%
	Defects discovered	1	2	1	0	7	11
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	6	17	8	4	22	57
11	**Field (Beta) testing**	14.00%	12.00%	14.00%	12.00%	**34.00%**	20.92%
	Defects discovered	1	2	1	1	7	12
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	6	15	6	4	14	45
12	**Acceptance testing**	13.00%	14.00%	15.00%	12.00%	**24.00%**	20.16%
	Defects discovered	1	2	1	0	6	10
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	5	13	6	3	8	35
	Test Defects Removed	**13**	**35**	**39**	**46**	**177**	**309**
	Testing Efficiency %	**73.96%**	**72.26%**	**87.63%**	**93.44%**	**95.45%**	**89.78%**
	Total Defects Removed	**245**	**987**	**1,144**	**1,297**	**442**	**4,114**
	Total Bad-fix injection	**7**	**30**	**34**	**39**	**13**	**123**
	Cumulative Removal %	**98.11%**	**98.68%**	**99.52%**	**99.75%**	**98.13%**	**99.13%**
	Remaining Defects	**5**	**13**	**6**	**3**	**8**	**36**
	High-severity Defects	**1**	**2**	**1**	**1**	**1**	**5**
	Security Defects	**0**	**0**	**0**	**0**	**0**	**1**
	Remaining Defects per Function Point	**0.0036**	**0.0102**	**0.0042**	**0.0025**	**0.0065**	**0.0278**
	Remaining Defects per K Function Points	**3.63**	**10.17**	**4.23**	**2.46**	**6.48**	**27.81**
	Remaining Defects per KLOC	**0.09**	**0.25**	**0.10**	**0.06**	**0.16**	**0.68**

Note: The letters "IV&V" in table 16 stand for "independent verification and validation." This is a method used by defense software projects but it seldom occurs in the civilian sector. The efficiency of IV&V is fairly low and the costs are fairly high.

DRE measures can be applied to any combination of pre-test and testing stages. Table 16 shows seven pre-test DRE activities and 12 kinds of testing: 19 forms of defect removal in total. This combination would only be used on large defense systems and also on critical medical devices. It might also be used on aircraft navigation and avionics packages. In other words software that might cause injury or death to humans if quality lags are the most likely to use both DRE measures and sophisticated combinations of pre-test and test removal methods.

As of 2016 the U.S. average for DRE is only about 92.50%. This is close to the average for Agile projects.

The U.S. norm is to use only static analysis before testing and six kinds of testing: unit test, function test, regression test, performance test, system test, and acceptance test. This combination usually results in about 92.50% DRE.

If static analysis is omitted and only six test stages are used, DRE is normally below 85%. In this situation quality problems are numerous.

Note that when a full suite of pre-test defect removal and test stages are used, the final number of defects released to customers often has more bugs originating in requirements and design than in code.

Due to static analysis and formal testing by certified test personnel, DRE for code defects can top 99.75%. It is harder to top 99% for requirements and design bugs since both resist testing and can only be found via inspections, or by text static analysis.

Software Quality and Software Security

Software quality and software security have a tight relationship. Security flaws are just another kind of defect potential. As defect potentials go up so do security flaws, as DRE declines more and more security flaws will be released.

Of course security has some special methods that are not part of traditional quality assurance. One of these is the use of ethical hackers and another is the use of penetration teams that deliberately try to penetrate the security defenses of critical software applications.

Security also includes social and physical topics that are not part of ordinary software operations. For example security requires careful vetting of personnel. Security for really critical applications may also require Faraday cages around computers to ensure that remote sensors are blocked and can't steal information from a distance or though building walls.

To provide an approximate set of values for high-severity defects and security flaws table 16 shows what happens when defect potentials increase and DRE declines. To add realism to this example table 17 uses a fixed size of 1000 function points. Delivered defects, high-severity defects, and security flaws are shown in whole numbers rather than defects per function point:

Table 17 Quality and Security Flaws for 1000 Function Points

Defect Potentials per FP	DRE	Delivered Defects per FP	Delivered Defects	High Severity Defects	Security Flaw Defects
2.50	99.50%	0.01	**13**	**1**	**0**
3.00	99.00%	0.03	**30**	**3**	**0**
3.50	97.00%	0.11	**105**	**10**	**1**
4.00	95.00%	0.20	**200**	**21**	**3**
4.25	**92.50%**	**0.32**	**319**	**35**	**4**
4.50	92.00%	0.36	**360**	**42**	**6**
5.00	87.00%	0.65	**650**	**84**	**12**
5.50	83.00%	0.94	**935**	**133**	**20**
6.00	78.00%	1.32	**1,320**	**206**	**34**

The central row in the middle of this table highlighted in **blue** show approximate 2016 U.S. averages in terms of delivered defects, high-severity defects, and latent security flaws for 1000 function points. The odds of a successful cyber-attack would probably be around 15%.

At the safe end of the spectrum where defect potentials are low and DRE tops 99% the number of latent security flaws is 0. The odds of a successful cyber-attack are very low at the safe end of the spectrum: probably below 1%.

At the dangerous end of the spectrum with high defect potentials and low DRE, latent security flaws top 20 for 1000 function points. This raises the odds of a successful cyber-attack to over 50%.

Software Quality and Technical Debt

Ward Cunningham introduced an interesting metaphor called “technical debt” which concerns latent defects present in software applications after deployment.

The idea of technical debt is appealing but unfortunately technical debt is somewhat ambiguous and every company tends to accumulate data using different methods so it is hard to get accurate

benchmarks.

In general technical debt deals with the direct costs of fixing latent defects as they are reported by users or uncovered by maintenance personnel. However there are other and larger costs associated with legacy software and also new software that are not included in technical debt:

1. Litigation against software outsource contractors or commercial software vendors by disgruntled users who sue for excessive defects.

2. Consequential damages or financial harm to users of defective software. For example if the computerized brake system of an automobile fails and causes a serious accident, neither the cost of repairing the auto nor any medical bills for injured passengers are included in technical debt.

3. Latent security flaws that are detected by unscrupulous organizations and lead to data theft, denial of service, or other forms of cyber-attack are not included in technical debt either.

Technical debt is an appealing metaphor but until consistent counting rules become available it is not a satisfactory quality metric. The author suggests that the really high cost topics of consequential damages, cyber-attacks, and litigation for poor quality should be included in technical debt or at least not ignored as they are in 2016.

Assume a software outsource vendor builds a 10,000 function point application for a client for a cost of $30,000,000 and it has enough bugs to make the client unhappy. True technical debt or the costs of repairing latent defects found and reported by clients over several years after deployment might cost about $5,000,000.

However depending upon what the application does, consequential damages to the client could top $25,000,000; litigation by the unhappy client might cost $5,000,000; severe cyber-attacks and data theft might cost $30,000,000: a total cost of $60,000,000 over and above the nominal amount for technical debt.

Of these problems cyber-attacks are the most obvious candidates to be added to technical debt because they are the direct result of latent security flaws present in the software when it was deployed. The main difference between normal bugs and security flaws is that cyber criminals can exploit security flaws to do very expensive damages to software (and even hardware) or to steal valuable and sometimes classified information.

In other words possible post-release costs due to poor quality control might approach or exceed twice the initial costs of development; and 12 times the costs of "technical debt" as it is normally calculated.

SNAP Metrics for Non-Functional Size

In 2011 the IFPUG organization developed a new metric for non-functional requirements. This metric is called "SNAP" which is sort of an acronym for "software non-functional assessment process." (No doubt future sociologists will puzzle over software naming conventions.)

Unfortunately the SNAP metric was not created to be equivalent to standard IFPUG function points. That means if you have 100 function points and 15 SNAP points you cannot add them together to create 115 total "points." This makes both productivity and quality studies more difficult because function point and SNAP work needs to be calculated separately.

Since one of the most useful purposes for function point metrics has been for predicting and measuring quality, the addition of SNAP metrics to the mix has raised the complexity of quality calculations.

Pasted below are the results of an experimental quality calculation tool developed by the author that can combine defect potentials and defect removal efficiency (DRE) for both function point metrics and the newer SNAP metrics.

Table 18: SNAP Software Defect Calculator

6/9/2016

l

Size in Function Points	**1,000**
Size in SNAP Points	**152**

Defect Origins	**Defects Per FP**	**Defects per SNAP**	**SNAP Percent**
Requirements	0.70	0.14	19.50%
Architecture	0.10	0.02	15.50%
Design	0.95	0.18	18.50%
Source code	1.15	0.13	11.50%
Security flaws	0.25	0.05	20.50%
Documents	0.45	0.02	3.50%
Bad Fixes	0.65	0.12	18.50%
TOTALS	**4.25**	**0.65**	**15.23%**

Defect	**Defect**	**Defects**

Origins	Potential	Potential
Requirements	700	21
Architecture	100	2
Design	950	27
Source code	1,150	20
Security flaws	250	8
Documents	450	2
Bad Fixes	650	18
TOTALS	**4,250**	**99**

Defect Origins	Removal Percent	Removal Percent
Requirements	75.00%	75.00%
Architecture	70.00%	70.00%
Design	96.00%	96.00%
Source code	98.00%	98.00%
Security flaws	87.00%	87.00%
Documents	95.00%	95.00%
Bad Fixes	78.00%	78.00%
Average	**85.57%**	**85.57%**

Defect Origins	Delivered Defects	Delivered Defects
Requirements	175	5
Architecture	30	1
Design	38	1
Source code	23	0
Security flaws	33	1
Documents	23	0
Bad Fixes	143	4
Total	**464**	**13**

Defect Origins	Delivered Per FP	Delivered per SNAP	SNAP Percent
Requirements	0.175	0.034	19.50%
Architecture	0.030	0.005	15.50%
Design	0.038	0.007	18.50%
Source code	0.023	0.003	11.50%

Security flaws	0.023	0.007	29.61%
Documents	0.143	0.026	18.50%
Bad Fixes	0.464	0.082	17.75%
Total	**0.896**	**0.164**	**18.31%**

In real life defect potentials go up with application size and defect removal efficiency (DRE) comes down with application size. This experimental tool holds defect potentials and DRE as constant values. The purpose is primarily to experiment with the ratios of SNAP defects and with DRE against SNAP bugs.

A great deal more study and more empirical data is needed before SNAP can actually become useful for software quality analysis. Right now there is hardly any empirical data available on SNAP and software quality.

Economic Value of High Software Quality

One of the major economic weaknesses of the software industry due to bad metrics and poor measurements is a total lack understanding of the economic value of high software quality. If achieving high quality levels added substantially to development schedules and development costs it might not be worthwhile to achieve it. But the good news is that high software quality levels comes with shorter schedules and lower costs than average or poor quality!

These reductions in schedules and costs, or course, are due to the fact that finding and fixing bugs has been the #1 software cost driver for over 50 years. When defect potentials are reduced and DRE is increased due to pre-test defect removal such as static analysis, then testing time and testing costs shrink dramatically.

Table 19 shows the approximate schedules in calendar months, the approximate effort in work hours per function point, and the approximate $ cost per function point that results from various combinations of software defect potentials and defect removal efficiency.

The good news for the software industry is that low defect potentials and high DRE levels are the fastest and cheapest way to build software applications!

Table 19: Schedules, Effort, Costs for 1000 Function Points
(Monthly costs = $10,000)

Defect Potentials per FP	DRE	Delivered Defects per FP	Delivered Defects	Schedule Months	Work Hours per Function Point	Development Cost per Function Point	$ per Defect (Caution!)
2.50	99.50%	0.01	**13**	13.34	12.00	**$909.09**	**$4,550.00**
3.00	99.00%	0.03	**30**	13.80	12.50	**$946.97**	**$3,913.00**
3.50	97.00%	0.11	**105**	14.79	13.30	**$1,007.58**	**$3,365.18**
4.00	95.00%	0.20	**200**	15.85	13.65	**$1,034.09**	**$2,894.05**
4.25	**92.50%**	**0.32**	**319**	**16.00**	**13.85**	**$1,050.00**	**$2,488.89**
4.50	92.00%	0.36	**360**	16.98	14.00	**$1,060.61**	**$2,140.44**
5.00	87.00%	0.65	**650**	18.20	15.00	**$1,136.36**	**$1,840.78**
5.50	83.00%	0.94	**935**	19.50	16.50	**$1,250.00**	**$1,583.07**
6.00	78.00%	1.32	**1,320**	20.89	17.00	**$1,287.88**	**$1,361.44**

The central row highlighted in **blue** shows approximate U.S. average values for 2016. This table also shows the "cost per defect" metric primarily to caution readers that this metric is inaccurate and distorts reality since it make buggy applications look cheaper than high-quality applications.

A Primer on Manufacturing Economics and the Impact of Fixed Costs

The reason for the distortion of the cost per defect metric is because cost per defect ignores the fixed costs of writing test cases, running test cases, and for maintenance the fact that the change team must be ready whether or not bugs are reported.

To illustrate the problems with the cost per defect metric, assume you have data on four identical applications of 1000 function points in size. Assume for all four that writing test cases costs $10,000 and running test cases costs $10,000 so fixed costs are $20,000 for all four cases.

Now assume that fixing bugs costs exactly $500 each for all four cases. Assume Case 1 found 100 bugs, Case 2 found 10 bugs, Case 3 found 1 bug, and Case 4 had zero defects with no bugs found by testing. Table 20 illustrates both cost per defect and cost per function point for these four cases:

Table 20: Comparison of $ per defect and $ per function point

	Case 1	Case 2	Case 3	Case 4
Fixed costs	$20,000	$20,000	$20,000	$20,000
Bug repairs	$50,000	$5,000	$500	$0
Total costs	$70,000	$25,000	$20,500	$20,000
Bugs found	100	10	1	0
$ per defect	**$700**	**$2,500**	**$20,500**	**Infinite**
$ per FP	**$70.00**	**$25.00**	**$20.50**	**$20.00**

As can be seen the "cost per defect" metric penalizes quality and gets more expensive as defect volumes decline. This is why hundreds of refereed papers all claim that cost per defect goes up later in development.

The real reason that cost per defect goes up is not that the actual cost of defect repairs goes up, but rather fixed costs make it look that way. Cost per function point shows the true economic value of high quality and this goes down as defects decline.

Recall a basic law of manufacturing economics that "*If a manufacturing process has a high percentage of fixed costs and there is a decline in the number of units produced, the cost per unit will go up*." For over 50 years the cost per defect metric has distorted reality and concealed the true economic value of high quality software.

Some researchers have suggested leaving out the fixed costs of writing and running test cases and only considering the variable costs of actual defect repairs. This violates both economic measurement principles and also and good sense.

Would you want a contractor to give you an estimate for building a house that only showed foundation and framing costs but not the more variable costs of plumbing, electrical wiring, and internal finishing? Software Cost of Quality (COQ) needs to include ALL of the cost elements of finding and fixing bugs and not just a small subset of those costs.

The author has read over 100 refereed software articles in major journals such as IEEE Transactions, IBM Systems Journal, Cutter Journal, and others that parroted the stock phrase "*It costs 100 times more to fix a bug after release than it does early in development*."

Not even one of these 100 articles identified the specific activities that were included in the cost per defect data. Did the authors include test case design, test case development, test execution, defect logging, defect analysis, inspections, desk checking, correctness proofs, static analysis, all forms of testing, post-release defects, abeyant defects, invalid defects, duplicate defects, bad fix injections, error-prone modules or any of the other topics that actually have a quantified impact on defect repairs?

Not even one of the 100 journal articles included such basic information on the work elements that comprised the "cost per defect" claims by the authors.

In medical journals this kind of parroting of a stock phrase without defining any of its elements would be viewed as professional malpractice. But the software literature is so lax and so used to bad data, bad metrics, and bad measures that none of the referees probably even noticed that the cost per defect claims were unsupported by any facts at all.

The omission of fixed costs also explains why "lines of code" metrics are invalid and penalize high-level languages. In the case of LOC metrics requirements, design, architecture, and other kinds of non-code work are fixed costs, so when there is a switch from a low-level language such as assembly to a higher level language such as Objective C the "cost per line of code" goes up.

Table 21 shows 15 programming languages with cost per function point and cost per line of code in side by side columns, to illustrate that LOC penalizes high-level programming languages, distorts reality, and reverses the true economic value of high-level programming languages:

Table 21: Productivity Expressed Using both LOC and Function Points

	Languages	Size in LOC	Coding Work hrs	Total Work hrs	Total Costs	$ per FP	$ per LOC
1	Application Generators	7,111	1,293	**4,293**	**$325,222**	**$325.22**	**$45.73**
2	Mathematica10	9,143	1,662	**4,662**	**$353,207**	**$353.21**	**$38.63**
3	Smalltalk	21,333	3,879	**6,879**	**$521,120**	**$521.12**	**$24.43**
4	Objective C	26,667	4,848	**7,848**	**$594,582**	**$594.58**	**$22.30**
5	Visual Basic	26,667	4,848	**7,848**	**$594,582**	**$594.58**	**$22.30**
6	APL	32,000	5,818	**8,818**	**$668,044**	**$668.04**	**$20.88**
7	Oracle	40,000	7,273	**10,273**	**$778,237**	**$778.24**	**$19.46**
8	Ruby	45,714	8,312	**11,312**	**$856,946**	**$856.95**	**$18.75**
9	Simula	45,714	8,312	**11,312**	**$856,946**	**$856.95**	**$18.75**
10	C#	51,200	9,309	**12,309**	**$932,507**	**$932.51**	**$18.21**
11	ABAP	80,000	14,545	**17,545**	**$1,329,201**	**$1,329.20**	**$16.62**
12	PL/I	80,000	14,545	**17,545**	**$1,329,201**	**$1,329.20**	**$16.62**
13	COBOL	106,667	19,394	**22,394**	**$1,696,511**	**$1,696.51**	**$15.90**
14	C	128,000	23,273	**26,273**	**$1,990,358**	**$1,990.36**	**$15.55**
15	Macro Assembly	213,333	38,788	**41,788**	**$3,165,748**	**$3,165.75**	**$14.84**

Recall that the standard economic definition for productivity for more than 200 years has been "*Goods or services produced per unit of labor or expense.*" If a line of code is selected as a unit of expense then moving to a high-level programming language will drive up the cost per LOC because of the fixed costs of non-code work.

Function point metrics, on the other hand, do not distort reality and are a good match to manufacturing economics and also to standard economics because they correctly show that the least expensive version has the highest economic productivity. LOC metrics make the most expensive version seem to have higher productivity than the cheapest, which of course violates standard economics.

Also, software has a total of 126 occupation groups. The only occupation that can be measured at with "lines of code" is that of programming. Function point metrics, on the other hand, can measure the productivity of non-code occupations such as business analysts, architects, data base designers, technical writers, project management and everybody else.

Table 22 illustrates the use of function points for 40 software development activities. It is obvious that serious software economic analysis needs to use activity-based costs and not just use single-point measures or phase-based measures neither of which can be validated.

Table 22: Function Points for Activity-Based Cost Analysis for 10,000 Function Points

	Development Activities	Work Hours per Funct. Pt.	Burdened Cost per Funct. Pt.	Project Cost	% of Total
1	Business analysis	0.01	$0.42	$4,200	0.02%
2	Risk analysis/sizing	0.00	$0.14	$1,400	0.01%
3	Risk solution planning	0.00	$0.21	$2,100	0.01%
4	Requirements	0.29	$23.33	$233,333	1.36%
5	Requirement. Inspection	0.24	$19.09	$190,909	1.11%
6	Prototyping	0.38	$30.00	$30,000	0.17%
7	Architecture	0.05	$4.20	$42,000	0.24%
8	Architecture. Inspection	0.04	$3.00	$30,000	0.17%
9	Project plans/estimates	0.04	$3.00	$30,000	0.17%
10	Initial Design	0.66	$52.50	$525,000	3.06%
11	Detail Design	0.88	$70.00	$700,000	4.08%
12	Design inspections	0.53	$42.00	$420,000	2.45%
13	Coding	6.60	$525.00	$5,250,000	30.58%
14	Code inspections	3.30	$262.50	$2,625,000	15.29%
15	Reuse acquisition	0.00	$0.14	$1,400	0.01%
16	Static analysis	0.01	$0.70	$7,000	0.04%
17	COTS Package purchase	0.01	$0.42	$4,200	0.02%

18	Open-source acquisition.	0.00	$0.21	$2,100	0.01%
19	Code security audit.	0.07	$5.25	$52,500	0.31%
20	Ind. Verif. & Valid. (IV&V)	0.01	$1.05	$10,500	0.06%
21	Configuration control.	0.03	$2.10	$21,000	0.12%
22	Integration	0.02	$1.75	$17,500	0.10%
23	User documentation	0.26	$21.00	$210,000	1.22%
24	Unit testing	1.06	$84.00	$840,000	4.89%
25	Function testing	0.94	$75.00	$750,000	4.37%
26	Regression testing	1.47	$116.67	$1,166,667	6.80%
27	Integration testing	1.06	$84.00	$840,000	4.89%
28	Performance testing	0.26	$21.00	$210,000	1.22%
29	Security testing	0.38	$30.00	$300,000	1.75%
30	Usability testing	0.22	$17.50	$175,000	1.02%
31	System testing	0.75	$60.00	$600,000	3.49%
32	Cloud testing	0.06	$4.38	$43,750	0.25%
33	Field (Beta) testing	0.03	$2.63	$26,250	0.12%
34	Acceptance testing	0.03	$2.10	$21,000	0.12%
35	Independent testing	0.02	$1.75	$17,500	0.10%
36	Quality assurance	0.18	$14.00	$140,000	0.82%
37	Installation/training	0.03	$2.63	$26,250	0.15%
38	Project measurement	0.01	$1.11	$11,053	0.06%
39	Project office	0.24	$19.09	$190,909	1.11%
40	Project management	1.76	$140.00	$1,400,000	8.15%
	Cumulative Results	**21.91**	**$1,743.08**	**$17,168,521**	**100.00%**

In table 22 the activities that are related to software quality are highlighted in blue. Out of a total of 40 activities 26 of them are directly related to quality. These 26 quality-related activities sum to 49.50% of software development costs while actual coding is only 30.58% of development costs.

So long as software is built using custom designs and manual coding defect detection and defect removal must be the major cost drivers of all software applications. Construction of software from certified reusable components would greatly increase software productivity and benefit the economics of not only software itself but of all industries that depend on software, which essentially means every industry in the world.

Table 22 shows the level of granularity needed to understand the cost structures of large software applications where coding is just over 30% of the total effort. Software management and C-level executives such as Chief Financial Officers (CFO) and Chief Information Officers (CIO) need to understand the complete set of activity-based costs and also costs by occupation group such as business analysts and architects over and above programmers.

When you build a house you need to know the costs of everything: foundations, framing, electrical systems, roofing, plumbing etc. You also need to know the separate costs of architects, carpenters, plumbers, electricians, and all of the other occupations that work on the house.

Here too for large systems in the 10,000 function point size range a proper understanding of software economics needs measurements of ALL activities and all occupation groups and not just coding programmers, whose effort is often less than 30% of the total effort for large systems.

Both LOC metrics and cost per defect metrics should probably be viewed as *professional malpractice* for software economic studies because they both distort reality and make bad results look better than good results.

It is no wonder that software progress resembles and drunkard's walk when hardly anybody knows how to measure either quality or productivity with metrics that make sense and match standard economics.

Software's Lack of Accurate Data and Education on Quality and Cost of Quality (COQ)

One would think that software manufacturing economics would be taught in colleges and universities as part of computer science and software engineering curricula, but universities are essentially silent on the topic of fixed costs probably because the software faculty does not understand software manufacturing economics either. There are a few exceptions such as the University of Montreal however.

The private software education companies and the professional associations are also silent on the topic of software economics and the hazards of cost per defect and lines of code. It is doubtful if either of these sectors understands software economics well enough to teach it. They certainly don't seem to understand either function points or quality metrics such as defect removal efficiency (DRE).

Even more surprising some of the major software consulting groups with offices and clients all over the world are also silent on software economics and the hazards of both cost per defect and lines of code. Gartner Group uses function points but apparently has not dealt with the impact of fixed costs and the distortions caused by the LOC and cost per defect metrics.

You would think that major software quality tool vendors such as those selling automated test tools, static analysis tools, defect tracking tools, automated correctness proofs, or test-case design methods based on cause-effect graphs or design of experiments would measure defect potentials and DRE because these metrics could help to demonstrate the value of their products. Recall that IBM used defect potentials and DRE metrics to prove the value of formal inspections back in 1973.

But the quality companies are just as clueless as their clients when it comes to defect potentials and defect removal efficiency (DRE) and the economic value of high quality. They make vast claims of quality improvements but provide zero quantitative data. For example only CAST Software that sells static analysis uses function points on a regular basis from among the major quality tool companies. But even CAST does not use defect potentials and DRE although some of their clients do.

You would also think that project management tool companies that market tools for progress and cost accumulation reporting and project dashboards would support function points and show useful economic metrics such as work hours per function point and cost per function point. You would also think they would support activity-based costs. However most project management tools do not support either function point metrics or activity-based costs, although a few do support earned value and some forms of activity-based cost analysis. This means that standard project management tools are not useful for software benchmarks since function points are the major benchmark metric.

The only companies and organizations that seem to know how to measure quality and economic productivity are the function point associations such as COSMIC, FISMA, IFPUG, and NESMA; the software benchmark organizations such as ISBSG, David's Consulting, Namcook Analytics, TIMetricas, Q/P Management Group, and several others; and some of the companies that sell parametric estimation tools such as KnowledgePlan, SEER, SLIM, and the author's Software Risk Master (SRM). In fact the author's SRM tool predicts software application size in a total of 23 metrics including all forms of function points plus story points, use-case points, physical and logical code, and a number of others. It even predicts bad metrics such as cost per defect and lines of code primarily to demonstrate to clients why those metrics distort reality.

Probably not one reader out of 1000 of this paper has quality and cost measures that are accurate enough to confirm or challenge the data in tables 19, 20, and 21 because software measures and metrics have been fundamentally incompetent for over 50 years. This kind of analysis can't be done with "cost per defect" or "lines of code" because they both distort reality and conceal the economic value of software quality.

However the comparatively few companies and fewer government organizations that do measure software costs and quality well using function points and DRE can confirm the results. The quality pioneers of Joseph Juran, W. Edwards Deming, and Phil Crosby showed that for manufactured products quality is not only free it also saves time and money.

The same findings are true for software, only software has lagged all other industries in discovering the economic value of high software quality because software metrics and measures have been so bad that they distorted reality and concealed progress.

The combination of function point metrics and defect removal efficiency (DRE) measures can finally prove that high software quality, like the quality of manufactured products, lowers

development costs and shortens development schedules. High quality also lowers maintenance costs, reduces the odds of successful cyber-attacks, and improves customer satisfaction levels.

Summary and Conclusions

The combination of *defect potentials* and *defect removal efficiency (DRE)* measures provide software engineering and quality personnel with powerful tools for predicting and measuring all forms of defect prevention and all forms of defect removal.

Function points are the best metric for normalizing software defect potentials because function points are the only metrics that can handle requirements, design, architecture, and other sources of non-code defects.

This paper uses IFPUG 4.3 function points. Other forms of function point metric such as COSMIC, FISMA, NESMA, etc. would be similar but not identical to the values shown here.

As of 2016 there is insufficient data on SNAP metrics to show defect potentials and defect removal efficiency. However it is suspected that non-functional requirements contribute to defect potentials in a significant fashion. There is insufficient data in 2016 to judge DRE values against non-functional defects.

Note that the author's Software Risk Master (SRM) tool predicts defect potentials and defect removal efficiency (DRE) as standard outputs for all projects estimated.

For additional information on 25 methods of pre-test defect removal and 25 forms of testing, see The Economics of Software Quality, Addison Wesley, 2012 by Capers Jones and Olivier Bonsignour.

References and Readings on Software Quality

Beck, Kent; Test-Driven Development; Addison Wesley, Boston, MA; 2002; ISBN 10: 0321146530; 240 pages.

Black, Rex; Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing; Wiley; 2009; ISBN-10 0470404159; 672 pages.

Chelf, Ben and Jetley, Raoul; "*Diagnosing Medical Device Software Defects Using Static Analysis*"; Coverity Technical Report, San Francisco, CA; 2008.

Chess, Brian and West, Jacob; Secure Programming with Static Analysis; Addison Wesley, Boston, MA; 20007; ISBN 13: 978-0321424778; 624 pages.

Cohen, Lou; Quality Function Deployment – How to Make QFD Work for You; Prentice Hall, Upper Saddle River, NJ; 1995; ISBN 10: 0201633302; 368 pages.

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

Everett, Gerald D. And McLeod, Raymond; Software Testing; John Wiley & Sons, Hoboken, NJ; 2007; ISBN 978-0-471-79371-7; 261 pages.

Gack, Gary; Managing the Black Hole: The Executives Guide to Software Project Risk; Business Expert Publishing, Thomson, GA; 2010; ISBN10: 1-935602-01-9.

Gack, Gary; *Applying Six Sigma to Software Implementation Projects*; http://software.isixsigma.com/library/content/c040915b.asp.

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

Hallowell, David L.; *Six Sigma Software Metrics, Part 1.*; http://software.isixsigma.com/library/content/03910a.asp.

International Organization for Standards; ISO 9000 / ISO 14000; http://www.iso.org/iso/en/iso9000-14000/index.html.

Jones, Capers: Software Risk Master (SRM) tutorial; Namcook Analytics LLC, Narragansett RI, 2015.

Jones, Capers: Software Defect Origins and Removal Methods; Namcook Analytics LLC; Narragansett RI, 2015.

Jones, Capers: The Mess of Software Metrics; Namcook Analytics LLC, Narragansett RI; 2015.

Jones, Capers; The Technical and Social History of Software Engineering; Addison Wesley, 2014.

Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley, Boston, MA; 2011; ISBN 978-0-13-258220-9; 587 pages.

Jones, Capers; Software Engineering Best Practices; McGraw Hill, New York; 2010; ISBN 978-0-07-162161-8;660 pages.

Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008; ISBN 978=0-07-150244-3; 662 pages.

Jones, Capers; Critical Problems in Software Measurement; Information Systems Management Group, 1993; ISBN 1-56909-000-9; 195 pages.

Jones, Capers; Software Productivity and Quality Today -- The Worldwide Perspective; Information Systems Management Group, 1993; ISBN -156909-001-7; 200 pages.

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

Jones, Capers; New Directions in Software Management; Information Systems Management Group; ISBN 1-56909-009-2; 150 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; 2nd edition; McGraw Hill, New York; 2007; 700 pages..

Jones, Capers; "The Economics of Object-Oriented Software"; SPR Technical Report; Software Productivity Research, Burlington, MA; April 1997; 22 pages.

Jones, Capers; "Software Project Management Practices: Failure Versus Success"; Crosstalk, October 2004.

Jones, Capers; "Software Estimating Methods for Large Projects"; Crosstalk, April 2005.

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.

Land, Susan K; Smith, Douglas B; Walz, John Z; Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards; WileyBlackwell; 2008; ISBN 10: 0470170808; 312 pages.

Mosley, Daniel J.; The Handbook of MIS Application Software Testing; Yourdon Press, Prentice Hall; Englewood Cliffs, NJ; 1993; ISBN 0-13-907007-9; 354 pages.

Myers, Glenford; The Art of Software Testing; John Wiley & Sons, New York; 1979; ISBN 0-471-04328-1; 177 pages.

Nandyal; Raghav; Making Sense of Software Quality Assurance; Tata McGraw Hill Publishing, New Delhi, India; 2007; ISBN 0-07-063378-9; 350 pages.

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

Royce, Walker E.; Software Project Management: A Unified Framework; Addison Wesley Longman, Reading, MA; 1998; ISBN 0-201-30958-0.

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