Software Economics and Function Point Metrics:

Thirty years of IFPUG Progress

Version 10.0 April 14, 2017

Capers Jones, Vice President and CTO, Namcook Analytics LLC
Web: www.Namcook.com
Email: Capers.Jones3@gmail.com

Keywords: IFPUG, cost per defect, economic productivity, function points, lines of code (LOC), manufacturing economics, software productivity, SNAP metrics, software metrics, software quality, technical debt.

Abstract

Calendar year 2017 marks the 30th anniversary of the International Function Point Users Group (IFPUG). This paper highlights some of the many modern uses of function point metrics. The software industry is one of the largest, wealthiest, and most important industries in the modern world. The software industry is also troubled by very poor quality and very high cost structures due to the expense of software development, maintenance, and endemic problems with poor quality control.

Accurate measurements of software development and maintenance costs and accurate measurement of quality would be extremely valuable. Function point metrics allow accurate measures.

Note: Many tables in this report are excerpts from the author's new 2017 series of three books with CRC Press: 1) *A Guide to Selecting Software Measures and Metrics;* 2) *A Quantified Comparison of 60 Software Development Methodologies*; 3) *Measuring and Comparing Global Software Productivity and Quality*.

Introduction

In the mid 1970's the author was commissioned by IBM executives to build IBM's first software estimation tool. In developing this tool we noted that "lines of code" was inaccurate for high-level languages. But I had no good solution at the time. What I did was convert LOC results into "equivalent assembly" lines of code and measured productivity using "equivalent assembly LOC per month." This worked mathematically but was an ugly and inelegant solution to the LOC problem.

Later in 1978 Al Albrecht and I both spoke at an IBM conference in Monterey, California. My talk was on the problems of lines of code metrics. Al's talk happened to be the first public speech on function points.

Al's team at IBM White Plans and the new function point metrics solved the LOC problem. Al and I became friends and later worked together. Soon after IFPUG was formed in Canada, and function point metrics began to advance on their path of becoming the #1 software metric.

IBM's Development of Function Point Metrics

The author was working at IBM in the 1960's and 1970's and was able to observe the origins of several IBM technologies such as inspections, parametric estimation tools, and function point metrics. This short paper discusses the origins and evolution of function point metrics.

In the 1960's and 1970's IBM was developing new programming languages such as APL, PL/I, PL/S etc. IBM executives wanted to attract customers to these new languages by showing clients higher productivity rates.

As it happens the compilers for various languages were identical in scope and had the same features. Some older compilers were coded in assembly language while newer compilers were coded in PL/S, which was a new IBM language for systems software.

When we measured the productivity of assembly-language compilers versus PL/S compilers using "lines of code" (LOC) we found that even though PL/S took less effort, the LOC metric of LOC per month favored assembly language.

This problem is easiest to see when comparing products that are almost identical but merely coded in different languages. Compilers, of course, are very similar. Other products besides compilers that are close enough in feature sets to have their productivity negatively impacted by LOC metrics are PBX switches, ATM banking controls, insurance claims handling, and sorts.

To show the value of higher-level languages the first IBM approach was to convert high-level languages into "*equivalent assembly language*." In other words we measured productivity against a synthetic size based on assembly language instead of against true LOC size in the actual higher level languages. This method was used by IBM from around 1968 through 1972.

An IBM vice president, Ted Climis, said that IBM was investing a lot of money into new and better programming languages. Neither he nor clients could understand why we had to use the old assembly language as the metric to show productivity gains for new languages. This was counter-productive to the IBM strategy of moving customers to better programming languages. He wanted a better metric that was language independent and could be used to show the value of all IBM high-level languages.

This led to the IBM investment in function point metrics and to the creation of a function-point development team under Al Albrecht at IBM White Plains.

Function Point metrics were developed by the IBM team by around 1975 and used internally and successfully. In 1978 IBM placed function point metrics in the public domain and announced them via a technical paper given by Al Albrecht at a joint IBM/SHARE/Guide conference in Monterey, California.

Table 1 shows the underlying reason for the IBM function point invention based on the early comparison of assembly language and PL/S for IBM compilers.

Table 1 shows productivity in four separate flavors:

1. Actual lines of code in the true languages.
2. Productivity based on "equivalent assembly code."
3. Productivity based on "function points per month."
4. Productivity based on "work hours per function point."

Note: table 1 uses simple round numbers to clarify the issues noted with LOC metrics.

Table 1: IBM Function Point Evolution Circa 1968-1975
(Results for two IBM compilers)

	Assembly Language	**PL/S Language**
Lines of code (LOC)	17,500.00	5,000.00
Months of effort	**30.00**	**12.50**
Hours of effort	3,960.00	1,650.00
LOC per month	**583.33**	**400.00**
Equivalent assembly	17,500.00	17,500.00
Equiv. Assembly/month	583.33	1,400.00

Function points	100.00	100.00
Function Points/month	**3.33**	**8.00**
Work hours per FP	39.60	16.50

The three rows highlighted in blue show the crux of the issue. LOC metrics tend to penalize high-level languages and make low-level languages such as assembly look better than they really are. Function points metrics, on the other hand, show tangible benefits from higher-level programming languages and this matches the actual expenditure of effort and standard economic analysis. Productivity of course is defined as "***goods or services produced per unit of labor or expense***."

The creation and evolution of function point metrics was based on a need to show IBM clients the value of IBM's emerging family of high-level programming languages such as PL/I, APL, and others.

This is still a valuable use of function points since there are more than 3,000 programming languages in 2016 and new languages are being created at a rate of more than one per month.

Another advantage of function point metrics vis a vis LOC metrics is that function points can measure the productivity of non-coding tasks such as creation of requirements and design documents. In fact function points can measure all software activities, while LOC can only measure coding.

Up until the explosion of higher-level programming languages occurred, assembly language was the only language used for systems software (the author programmed in assembly for several years when starting out as a young programmer).

With only one programming language LOC metrics worked reasonably well. It was only when higher-level programming languages appeared that the LOC problems became apparent. It was soon realized that the essential problem with the LOC metric is really nothing more than a basic issue of manufacturing economics that had been understood by other industries for over 200 years.

This is a fundamental law of manufacturing economics: "***When 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.***"

The software non-coding work of requirements, design, and documentation act like fixed costs. When there is a move from a low-level language such as assembly to a higher-level language such as PL/S, the cost per unit will go up, assuming that LOC is the "unit" selected for

measuring the product. This is because of the fixed costs of the non-code work and the reduction of code "units" for higher-level programming languages.

Function point metrics are not based on code at all, but are an abstract metric that defines the essence of the features that the software provides to users. This means that applications with the same feature sets will be the same size in terms of function points no matter what languages they are coded in. Productivity and quality can go up and down, of course, but they change in response to team skills.

Once function points were released by IBM in 1978 other companies began to use them, and soon the International Function Point User's Group (IFPUG) was formed in Canada.

Today in 2017 there are hundreds of thousands of function point users and hundreds of thousands of benchmarks based on function points. In 1987 the International Function Point User's Group (IFPUG) was first formed in Canada. Today IFPUG has become the largest software measurement organization in the world.

Today there are also several other varieties of function points such as COSMIC, FISMA, NESMA, etc. IFPUG is the major form of function point metrics in the United States; the other forms are used elsewhere.

Overall function points have proven to be a successful metric and are now widely used for productivity studies, quality studies, and economic analysis of software trends. Function point metrics are supported by parametric estimation tools and also by benchmark studies. There are also several flavors of automatic function point tools. There are also function point associations in most industrialized countries. There are also ISO standards for functional size measurement.

(There was never an ISO standard for code counting and counting methods vary widely from company to company and project to project. In a benchmark study performed for a "LOC" shop we found four sets of counting rules for LOC that varied by over 500%.)

Table 2 shows countries with increasing function point usage circa 2017, and it also shows the countries where function point metrics are now required for government software projects.

Table 2: Countries Expanding Use of Function Points 2017

1	Argentina	
2	Australia	
3	Belgium	
4	**Brazil**	**Required for government contracts 2008**
5	Canada	
6	China	
7	Finland	
8	France	

9	Germany	
10	India	
11	**Italy**	**Required for government contracts 2012**
12	**Japan**	**Required for government contracts 2014**
13	**Malaysia**	**Required for government contracts 2015**
14	Mexico	
15	Norway	
16	Peru	
17	Poland	
18	Singapore	
19	**South Korea**	**Required for government contracts 2014**
20	Spain	
21	Switzerland	
22	Taiwan	
23	The Netherlands	
24	United Kingdom	
25	United States	

Several other countries will probably also mandate function points for government software contracts by 2017. Poland may be next since their government is discussing function points for contracts. Eventually most countries will do this.

In retrospect function point metrics have proven to be a powerful tool for software economic and quality analysis.

The software industry has become one of the largest and most successful industries in history. However software applications are among the most expensive and error-prone manufactured objects in history.

Software Historical Measurement Problems

Software needs a careful analysis of economic factors and much better quality control than is normally accomplished. In order to achieve these goals, software also needs accurate and reliable metrics and good measurement practices. Unfortunately the software industry lacks both circa 2017.

This paper deals with some of the most glaring problems of software metrics and suggests a metrics and measurement suite that can actually explore software economics and software quality with precision. The suggested metrics can be predicted prior to development and then measured after release.

Following are descriptions of the more common software metric topics in alphabetical order:

Backfiring is a term that refers to mathematical conversion between lines of code and function points. This method was first developed by A.J. Albrecht and colleagues during the original creation of function point metrics, since the IBM team had LOC data for the projects they used for function points. IBM used logical code statements for backfiring rather than physical LOC. There are no ISO standards for backfiring. Backfiring is highly ambiguous and varies by over 500% from language to language and company to company. A sample of "backfiring" is the ratio of about 106.7 statements in the procedure and data divisions of COBOL for one IFPUG function point. Consulting companies sell tables of backfire ratios for over 1000 languages, but the tables are not the same from vendor to vendor. Backfiring is not endorsed by any of the function point associations. Yet probably as many as 100,000 software projects have used backfiring because it is quick and inexpensive, even though very inaccurate with huge variances from language to language and programmer to programmer.

Benchmarks in a software context often refer to the effort and costs for developing an application. Benchmarks are expressed in a variety of metrics such as "work hours per function point," "function points per month," "lines of code per month," "work hours per KLOC," "story points per month," and many more. Benchmarks also vary in scope and range from project values, phase values, activity values, and task values. There are no ISO standards for benchmark contents. Worse, many benchmarks "leak" and omit over 50% of true software effort. The popular benchmark of "design, code, and unit test" termed DCUT contains only about 30% of total software effort. The most common omissions from benchmarks include unpaid overtime, management, and the work of part-time specialists such as technical writers and software quality assurance. Thus benchmarks from various sources such as ISBSG, QSM, and others cannot be directly compared since they do not contain the same information. The best and most reliable benchmarks feature activity-based costs and include the full set of development tasks; i.e. requirements, architecture, business analysis, design, coding, testing, quality assurance, documentation, project management, etc.

Cost estimating for software projects is generally inaccurate and usually optimistic. About 85% of projects circa 2017 use inaccurate manual estimates. The other 15% use the more accurate parametric estimating tools of which these are the most common estimating tools in 2015, shown in alphabetical order: COCOMO, COCOMO clones, CostXpert, ExcelerPlan, KnowledgePlan, SEER, SLIM, Software Risk Master (SRM), and TruePrice. A study by the author that compared 50 manual estimates against 50 parametric estimates found that only 4 of the 50 manual estimates were within plus or minus 5% and the average was 34% optimistic for costs and 27% optimistic for schedules. For manual estimates, the larger the projects the more optimistic the results. By contrast 32 of the 50 parametric estimates were within plus or minus 5% and the deviations for the others averaged about 12% higher for costs and 6% longer for schedules. Conservatism is the "fail safe" mode for estimates. The author's SRM tool has a patent-pending early sizing feature based on pattern matching that allows it to be used 30 to 180 days earlier than the other parametric estimation tools. It also predicts topics not included in the

others such as litigation risks, costs of breach of contract litigation for the plaintiff and defendant, and document sizes and costs for 20 key document types such as requirements, design, user manuals, plans, and others. The patent-pending early sizing feature of SRM produces size in a total of 23 metrics including function points, story points, use case points, logical code statements, physical lines of code, and many others.

Cost per defect metrics penalize quality and makes the buggiest software look cheapest. There are no ISO or other standards for calculating cost per defect. Cost per defect does not measure the economic value of software quality. The urban legend that it costs 100 times as much to fix post-release defects as early defects is not true and is based on ignoring fixed costs. Due to fixed costs of writing and running test cases, cost per defect rises steadily because fewer and fewer defects are found. This is caused by a standard rule of manufacturing economics: "*if a process has a high percentage of fixed costs and there is a reduction in the units produced, the cost per unit will go up*." This explains why cost per defects seems to go up over time even though actual defect repair costs are flat and do not change very much. There are of course very troubling defects that are expensive and time consuming, but these are comparatively rare. Appendix A explains the problems of cost per defect metrics.

Defect removal efficiency (DRE) was developed by IBM circa 1970. The original IBM version of DRE measured internal defects found by developers and compared them to external defects found by clients in the first 90 days following release. If developers found 90 bugs and clients reported 10 bugs, DRE is 90%. This measure has been in continuous use by hundreds of companies since about 1975. However there are no ISO standards for DRE. The International Software Benchmark Standards Group (ISBSG) unilaterally changed the post-release interval to 30 days in spite of the fact that the literature on DRE since the 1970's was based on a 90 day time span, such as the author's 1991 version of Applied Software Measurement and his more recent book on The Economics of Software Quality with Olivier Bonsignour. Those with experience in defects and quality tracking can state with certainty that a 30 day time window is too short; major applications sometimes need more than 30 days of preliminary installation and training before they are actually used. Of course bugs will be found long after 90 days; but experience indicates that a 90-day interval is sufficient to judge the quality of software applications. A 30 day interval is not sufficient.

Earned value management (EVM) is a method of combining schedule, progress, and scope. It originated in the 1960's for government contracts and has since been applied to software with reasonable success. Although earned value is relatively successful, it really needs some extensions to be a good fit for software projects. The most urgent extension would be to link progress to quality and defect removal. Finding and fixing bugs is the most expensive software activity. It would be easy to include defect predictions and defect removal progress into the earned value concept. Another extension for software would be to include the specific documents that are needed for large software applications. If the earned-value approach included quality topics, it would be very useful for contracts and software outsource agreements. EVM is

in use for defense software contracts, but the omission of quality is a serious problem since finding and fixing bugs is the most expensive single cost driver for software. The U.S. government requires earned value for many contracts. The governments of Brazil and South Korea require function points for software contracts. Most projects that end up in court for breach of contract do so because of poor quality. It is obvious that combining earned-value metrics, defect and quality metrics, and function point metrics would be a natural fit to all software contracts and would probably lead to fewer failures and better overall performance.

Defect density metrics measure the number of bugs released to clients. There are no ISO or other standards for calculating defect density. One method counts only code defects released. A more complete method used by the author includes bugs originating in requirements, architecture, design, and documents as well as code defects. The author's method also includes "bad fixes" or bugs in defect repairs themselves. There is more than a 500% variation between counting only released code bugs and counting bugs from all sources. For example requirements defects comprise about 20% of released software problem reports.

Function point metrics were invented by IBM circa 1975 and placed in the public domain circa 1978. Function point metrics do measure economic productivity using both "*work hours per function point*" and "*function points per month*". They also are useful for normalizing quality data such as "defects per function point". However there are numerous function point variations and they all produce different results: Automatic, backfired, COSMIC, Fast, FISMA, IFPUG, Mark II, NESMA, Unadjusted, etc. There are ISO standards for COSMIC, FISMA, IFPUG, and NESMA. However in spite of ISO standards all four produce different counts. Adherents of each function point variant claim "accuracy" as a virtue but there is no cesium atom or independent way to ascertain accuracy so these claims are false. For example COSMIC function points produce higher counts than IFPUG function points for many applications but that does not indicate "accuracy" since there is no objective way to know accuracy.

Goal/Question metrics (GQM) were invented by Dr. Victor Basili of the University of Maryland. The concept is appealing. The idea is to specify some kind of tangible goal or target, and then think of questions that must be answered to achieve the goal. This is a good concept for all science and engineering and not just software. However, since every company and project tends to specify unique goals the GQM method does not lend itself to either parametric estimation tools or to benchmark data collection. It would not be difficult to meld GQM with function point metrics and other effective software metrics such as defect removal efficiency (DRE). For example several useful goals might be "*How can we achieve defect potentials of less than 1.0 per function point?*" or "*How can we achieve productivity rates of 100 function points per month?*" Another good goal which should actually be a target for every company and every software project in the world would be "*How can we achieve more than 99% in defect removal efficiency (DRE)?*"

ISO/IEC standards are numerous and cover every industry; not just software. However these standards are issued without any proof of efficacy. After release some standards have proven to be useful, some are not so useful, and a few are being criticized so severely that some software consultants and managers are urging a recall such as the proposed ISO/IEC testing standard. ISO stands for the International Organization for Standards (in French) and IEC stands for International Electrical Commission. While ISO/IEC standards are the best known, there are other standards groups such as the Object Management Group (OMG) which recently published a standard on automatic function points. Here too there is no proof of efficacy prior to release. There are also national standards such as ANSI or the American National Standards Institute, and also military standards by the U.S. Department of Defense (DoD) and by similar organizations elsewhere. The entire topic of standards is in urgent need of due diligence and of empirical data that demonstrates the value of specific standards after issuance. In total there are probably several hundred standards groups in the world with a combined issuance of over 1000 standards, of which probably 50 apply to aspects of software. Of these only a few have solid empirical data that demonstrates value and efficacy.

Lines of code (LOC) metrics penalize high-level languages and make low-level languages look better than they are. LOC metrics also make requirements and design invisible. There are no ISO or other standards for counting LOC metrics. About half of the papers and journal articles use physical LOC and half use logical LOC. The difference between counts of physical and logical LOC can top 500%. The overall variability of LOC metrics has reached an astounding 2,200% as measured by Joe Schofield, the former president of IFPUG! LOC metrics make requirements and design invisible and also ignore requirements and design defects, which outnumber code defects. Although there are benchmarks based on LOC, the intrinsic errors of LOC metrics make them unreliable. Due to lack of standards for counting LOC, benchmarks from different vendors for the same applications can contain widely different results. Appendix B provides a mathematical proof that LOC metrics do not measure economic productivity by showing 79 programming languages with function points and LOC in a side-by-side format.

SNAP point metrics are a new variation on function points introduced by IFPUG in 2012. The term SNAP is an acronym for "software non-functional assessment process." The basic idea is that software requirements have two flavors: 1) functional requirements needed by users; 2) non-functional requirements due to laws, mandates, or physical factors such as storage limits or performance criteria. The SNAP committee view is that these non-functional requirements should be sized, estimated, and measured separately from function point metrics. Thus SNAP and function point metrics are not additive, although they could have been. Having two separate metrics for economic studies is awkward at best and inconsistent with other industries. For that matter it seems inconsistent with standard economic analysis in every industry. Almost every industry has a single normalizing metric such as "cost per square foot" for home construction or "cost per gallon" for gasoline and diesel oil. As of 2017 none of the parametric estimation tools have fully integrated SNAP and it may be that they won't since the costs of adding SNAP are

painfully expensive. As a rule of thumb non-functional requirements are about equal to 15% of functional requirements, although the range is very wide. The author's parametric tool calculates SNAP points but adds the effort for non-functional requirements to the total effort for the entire project, so net productivity is expressed in terms of cost per function point.

Story point metrics are widely used for agile projects with "user stories." Story points have no ISO standard for counting or any other standard. They are highly ambiguous and vary by as much as 400% from company to company and project to project. There are few useful benchmarks using story points. Obviously story points can't be used for projects that don't utilize user stories so they are worthless for comparisons against other design methods.

Technical debt is a new metric and rapidly spreading. It is a brilliant metaphor developed by Ward Cunningham. The concept of "technical debt" is that topics deferred during development in the interest of schedule speed will cost more after release than they would have cost initially. However there are no ISO standards for technical debt and the concept is highly ambiguous. It can vary by over 500% from company to company and project to project. Worse, technical debt does not include all of the costs associated with poor quality and development short cuts. Technical debt omits canceled projects, consequential damages or harm to users, and the costs of litigation for poor quality.

Use case points are used by projects with designs based on "use cases" which often utilize IBM's Rational Unified Process (RUP). There are no ISO standards for use cases. Use cases are ambiguous and vary by over 200% from company to company and project to project. Obviously use cases are worthless for measuring projects that don't utilize use cases, so they have very little benchmark data. This is yet another attempt to imitate the virtues of function point metrics, only with somewhat less rigor and with imperfect counting rules as of 2015.

Velocity is an agile metric that is used for prediction of sprint and project outcomes. It uses historical data on completion of past work units combined with the assumption that future work units will be about the same. Of course it is necessary to know future work units for the method to operate. The concept of velocity is basically similar to the concept of using historical benchmarks for estimating future results. However as of 2015 velocity has no ISO standards and no certification. There are no standard work units and these can be story points or other metrics such as function points or use case points, or even synthetic concepts such as "days per task." If agile projects use function points then they could gain access to large volumes of historical data using activity-based costs; i.e. requirements effort, design effort, code effort, test effort, integration effort, documentation effort, etc. Story points have too wide a range of variability from company to company and project to project; function points are much more consistent across various kinds of projects. Of course COSMIC, IFPUG, and the other variants don't have exactly the same results.

Defining Software Productivity

For more than 200 years the standard economic definition of productivity has been, "***Goods or services produced per unit of labor or expense***." This definition is used in all industries, but has been hard to use in the software industry. For software there is ambiguity in what constitutes our "*goods or services*."

The oldest unit for software "goods" was a "*line of code*" or LOC. More recently software goods have been defined as "*function points*." Even more recent definitions of goods include "*story points*" and "*use case points*." The pros and cons of these units have been discussed and some will be illustrated in the appendices.

Another important topic taken from manufacturing economics has a big impact on software productivity that is not yet well understood even in 2017: fixed costs.

A basic law of manufacturing economics that is valid for all industries including software is the following: "***When a development 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***."

When a "*line of code*" is selected as the manufacturing unit and there is a switch from a low-level language such as assembly to a high level language such as Java, there will be a reduction in the number of units developed.

But the non-code tasks of requirements and design act like fixed costs. Therefore the cost per line of code will go up for high-level languages. This means that LOC is not a valid metric for measuring economic productivity as proven in Appendix B.

For software there are two definitions of productivity that match standard economic concepts:

1. *Producing a specific quantity of deliverable units for the lowest number of work hours.*
2. *Producing the largest number of deliverable units in a standard work period such as an hour, month, or year.*

In definition 1 deliverable goods are constant and work hours are variable.

In definition 2 deliverable goods are variable and work periods are constant.

The common metrics "*work hours per function point*" and "*work hours per KLOC*" are good examples of productivity definition 1.

The metrics "*function points per month*" and "*lines of code per month*" are examples of definition 2.

However for "lines of code" the fixed costs of requirements and design will cause apparent productivity to be reversed, with low-level languages seeming better than high-level languages, as shown by the 79 languages listed in Appendix B.

Definition 2 will also encounter the fact that the number of work hours per month varies widely from country to country. For example India works 190 hours per month while the Netherlands work only 115 hours per month. This means that productivity definitions 1 and 2 will not be the same. A given number of work hours would take fewer calendar months in India than in the Netherlands due to the larger number of monthly work hours.

Table 3 shows the differences between "*work hours per function point"* and "*function points per month"* for 52 countries. The national work hour column is from the Organization of International Cooperation and Development (OECD). Table 1 assumes a constant value of 15 work hours per function point for an identical application in every country shown.

Table 3: Comparison of Work Hours per FP and FP per Month

		OECD National Work hours per month	Work Hours per Function Point	Function Points per Month
1	India	190.00	15.00	13.47
2	Taiwan	188.00	15.00	13.20
3	Mexico	185.50	15.00	13.17
4	China	186.00	15.00	12.93
5	Peru	184.00	15.00	12.67
6	Colombia	176.00	15.00	12.13
7	Pakistan	176.00	15.00	12.13
8	Hong Kong	190.00	15.00	12.01
9	Thailand	168.00	15.00	11.73
10	Malaysia	192.00	15.00	11.73
11	Greece	169.50	15.00	11.70
12	South Africa	168.00	15.00	11.60
13	Israel	159.17	15.00	11.14
14	Viet Nam	160.00	15.00	11.07
15	Philippines	160.00	15.00	10.93
16	Singapore	176.00	15.00	10.92
17	Hungary	163.00	15.00	10.87
18	Poland	160.75	15.00	10.85
19	Turkey	156.42	15.00	10.69
20	Brazil	176.00	15.00	10.65

21	Panama	176.00	15.00	10.65
22	Chile	169.08	15.00	10.51
23	Estonia	157.42	15.00	10.49
24	Japan	145.42	15.00	10.49
25	Switzerland	168.00	15.00	10.45
26	Czech Republic	150.00	15.00	10.00
27	Russia	164.42	15.00	9.97
28	Argentina	168.00	15.00	9.91
29	Korea - South	138.00	15.00	9.60
30	**United States**	**149.17**	15.00	**9.47**
31	Saudi Arabia	160.00	15.00	9.44
32	Portugal	140.92	15.00	9.39
33	United Kingdom	137.83	15.00	9.32
34	Finland	139.33	15.00	9.29
35	Ukraine	156.00	15.00	9.20
36	Venezuela	152.00	15.00	9.10
37	Austria	134.08	15.00	8.94
38	Luxembourg	134.08	15.00	8.94
39	Italy	146.00	15.00	8.75
40	Belgium	131.17	15.00	8.74
41	New Zealand	144.92	15.00	8.68
42	Denmark	128.83	15.00	8.59
43	Canada	142.50	15.00	8.54
44	Australia	144.00	15.00	8.50
45	Ireland	127.42	15.00	8.49
46	Spain	140.50	15.00	8.42
47	France	123.25	15.00	8.22
48	Iceland	142.17	15.00	8.00
49	Sweden	135.08	15.00	7.97
50	Norway	118.33	15.00	7.89
51	Germany	116.42	15.00	7.76
52	Netherlands	115.08	15.00	7.67
	Average	**155.38**	**15.00**	**10.13**

No one to date has produced a table similar to table 1 for SNAP metrics but it is obvious that work hours per SNAP point and SNAP points per month will follow the same global patterns as do the older function point metrics.

Of course differences in experience, methodologies, languages, and other variables also impact both forms of productivity. The point of table 1 is that the two forms are not identical from country to country due to variations in local work patterns.

Defining Software Quality

As we all know the topic of "*quality*" is somewhat ambiguous in every industry. Definitions for quality can encompass subjective aesthetic quality and also precise quantitative units such as numbers of defects and their severity levels.

Over the years software has tried a number of alternate definitions for quality that are not actually useful. For example one definition for software quality has been "*conformance to requirements*."

Requirements themselves are filled with bugs or errors that comprise about 20% of the overall defects found in software applications. Defining quality as conformance to a major source of errors is circular reasoning and clearly invalid. We need to include requirements errors in our definition of quality.

Another definition for quality has been "*fitness for use*." But this definition is ambiguous and cannot be predicted before the software is released, or even measured well after release.

It is obvious that a workable definition for software quality must be unambiguous and capable of being predicted before release and then measured after release and should also be quantified and not purely subjective.

Another definition for software quality has been a string of words ending in "…ility" such as reliability and maintainability. However laudable these attributes are, they are all ambiguous and difficult to measure. Further, they are hard to predict before applications are built.

The quality standard ISO/IEC 9126 includes a list of words such as portability, maintainability, reliability, and maintainability. It is astonishing that there is no discussion of defects or bugs. Worse, the ISO/IEC definitions are almost impossible to predict before development and are not easy to measure after release nor are they quantified. It is obvious that an effective quality measure needs to be predictable, measurable, and quantifiable.

Reliability is predictable in terms of mean time to failure (MTTF) and mean time between failures (MTBF). Indeed these are standard predictions from the author's Software Risk Master (SRM) tool. However reliability is inversely proportional to delivered defects. Therefore the ISO quality standards should have included defect potentials, defect removal efficiency (DRE), and delivered defect densities.

An effective definition for software quality that can be both predicted before applications are built and then measured after applications are delivered is: "***Software quality is the absence of defects which would either cause the application to stop working, or cause it to produce incorrect results***."

Because delivered defects impact reliability, maintainability, usability, fitness for use, conformance to requirements, and also customer satisfaction any effective definition of software quality must recognize the central importance of achieving low volumes of delivered defects. Software quality is impossible without low levels of delivered defects no matter what definition is used.

This definition has the advantage of being applicable to all software deliverables including requirements, architecture, design, code, documents, and even test cases.

If software quality focuses on the prevention or elimination of defects, there are some effective corollary metrics that are quite useful.

The "***defect potential***" of a software application is defined as the sum total of bugs or defects that are likely to be found in requirements, architecture, design, source code, documents, and "bad fixes" or secondary bugs found in bug repairs themselves. The "defect potential" metric originated in IBM circa 1973 and is fairly widely used among technology companies.

The "***defect detection efficiency***" **(DDE)** is the percentage of bugs found prior to release of the software to customers.

The "***defect removal efficiency***" **(DRE)** is the percentage of bugs found and repaired prior to release of the software to customers.

DDE and DRE were developed in IBM circa 1973 but are widely used by technology companies in every country. As of 2015 the average DRE for the United States is just over 90%.

(DRE is normally measured by comparing internal bugs against customer reported bugs for the first 90 days of use. If developers found 90 bugs and users reported 10 bugs, the total is 100 bugs and DRE would be 90%.)

Another corollary metric is that of "***defect severity.***" This is a very old metric dating back to IBM in the early 1960's. IBM uses four severity levels:

- Severity 1 Software is inoperable < 1%
- Severity 2 Major feature disabled or incorrect < 15%
- Severity 3 Minor error; software is usable < 40%
- Severity 4 Cosmetic error that does not affect results < 35%

To clarify these various terms, table 4 shows defect potentials, and DRE for an application of 1000 function points coded in the Java language using Agile development. (Table 2 uses even numbers to simplify the math. The author's Software Risk Master (SRM) tool predicts the same kinds of values for actual projects.).

Table 4: Software Quality for 1000 Function Points, Java, and Agile Development

Defect Potentials	**Number of Bugs**	**Defects Per FP**
Requirements defects	750	0.75
Architecture defects	150	0.15
Design defects	1,000	1.00
Code defects	1,350	1.35
Document defects	250	0.25
Sub Total	**3,500**	**3.50**
Bad fixes	150	**0.15**
TOTAL	**3,650**	**3.65**
Defect removal Efficiency (DRE)	**97.00%**	**97.00%**
Defects removed	3,540	3.54
Defects delivered	**110**	**0.11**
High-severity delivered	**15**	**0.02**

All of the values shown in Table 4 can be predicted before applications are developed and then measured after the applications are released. Thus software quality can move from an ambiguous and subjective term to a rigorous and quantitative set of measures that can even be included in software contracts. Note that bugs from requirements and design cannot be quantified using lines of code or KLOC, which is why function points are the best choice for quality measurements. It is possible to retrofit LOC after the fact, but in real life LOC is not used for requirements, architecture, and design bug predictions.

Note that table 4 combines non-functional and functional requirements defects, which might be separate categories if SNAP metrics are used. However in almost 100% of software requirements documents studied by the author functional and non-functional requirements are both combined without any distinction in the requirements themselves.

Patterns of Successful Software Measurements and Metrics

Since the majority of global software projects are either not measured at all, only partially measured, or measured with metrics that violate standard economic assumptions, what does work? Following are discussions of the most successful combinations of software metrics available today in 2017.

Successful Software Measurement and Metric Patterns

1. Function points for normalizing productivity data
2. Function points for normalizing quality data
3. SNAP metrics for non-functional requirements
4. Defect potentials based on all defect types
5. Defect removal efficiency (DRE) based on all defect types
6. Defect removal efficiency (DRE) including inspections and static analysis
7. Defect removal efficiency (DRE) based on a 90-day post release period
8. Activity-based benchmarks for development
9. Activity-based benchmarks for maintenance
10. Cost of quality (COQ) for quality economics
11. Total cost of ownership (TCO) for software economics

Let us consider these 11 patterns of successful metrics.

Function points for normalizing productivity data

It is obvious that software projects are built by a variety of occupations and use a variety of activities including

1. Requirements
2. Design
3. Coding
4. Testing
5. Integration
6. Documentation
7. Management

The older lines of code (LOC) metric is worthless for estimating or measuring non-code work. Function points can measure every activity individually and also the combined aggregate totals of all activities.

Note that the new SNAP metric for non-functional requirements is not included. Integrating SNAP into productivity and quality predictions and measurements is still a work in progress. Future versions of this paper will discuss SNAP.

Function Points for Normalizing Software Quality

It is obvious that software bugs or defects originate in a variety of sources including but not limited to:

1. Requirements defects
2. Architecture defects
3. Design defects
4. Coding defects
5. Document defects
6. Bad fixes or defects in bug repairs

The older lines of code metric is worthless for estimating or measuring non-code defects but function points can measure every defect source.

Defect Potentials Based on all Defect Types

The term "defect potential" originated in IBM circa 1965 and refers to the sum total of defects in software projects that originate in requirements, architecture, design, code, documents, and "bad fixes" or bugs in defect repairs. The older LOC metric only measures code defects, and they are only a small fraction of total defects. The current distribution of defects for an application of 1000 function points in Java is approximately as follows:

Defect Sources	**Defects per function point**
Requirements defects	0.75
Architecture defects	0.15
Design defects	1.00
Code defects	1.25
Document defects	0.20
Bad fix defects	0.15
Total Defect Potential	**3.65**

There are of course wide variations based on team skills, methodologies, CMMI levels, programming languages, and other variable factors.

Defect Removal Efficiency (DRE) Based on All Defect Types

Since requirements, architecture, and design defects outnumber code defects, it is obvious that measures of defect removal efficiency (DRE) need to include all defect sources. It is also obvious to those who measure quality that getting rid of code defects is easier than getting rid of

other sources. Following are representative values for defect removal efficiency (DRE) by defect source for an application of 1000 function points in the C programming language:

Defect Sources	Defect Potential	DRE Percent	Delivered Defects
Requirements defects	1.00	85.00%	0.15
Architecture defects	0.25	75.00%	0.06
Design defects	1.25	90.00%	0.13
Code defects	1.50	97.00%	0.05
Document defects	0.50	95.00%	0.03
Bad fix defects	0.50	80.00%	0.10
Totals	**5.00**	**89.80%**	**0.51**

As can be seen DRE against code defects is higher than against other defect sources. But the main point is that only function point metrics can measure and include all defect sources. The older LOC metric is worthless for requirements, design, and architecture defects.

Defect Removal Efficiency Including Inspections and Static Analysis

Serious study of software quality obviously needs to include pre-test inspections and static analysis as well as coding.

The software industry has concentrated only on code defects and only on testing. This is short sighted and insufficient. The software industry needs to understand all defect sources and every form of defect removal including pre-test inspections and static analysis. The approximate defect removal efficiency levels (DRE) of various defect removal stages are shown below:

Table 5: Software Defect Potentials and Defect Removal Efficiency (DRE)

Note 1: The table represents high quality defect removal operations.
Note 2: The table illustrates calculations from Software Risk Master ™ (SRM)

Application type	**Embedded**
Application size in function points	**1,000**
Application language	Java
Language level	6.00
Source lines per FP	53.33
Source lines of code	53,333
KLOC of code	53.33

PRE-TEST DEFECT REMOVAL ACTIVITIES

	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 FP	**0.35**	**0.97**	**1.19**	**1.47**	**0.18**	**4.16**
	Defect potentials	**355**	**966**	**1,189**	**1,469**	**184**	**4,163**
1	**Requirement inspection**	5.00%	**87.00%**	10.00%	5.00%	8.50%	25.61%
	Defects discovered	18	840	119	73	16	1,066
	Bad-fix injection	1	25	4	2	0	32
	Defects remaining	337	100	1,066	1,394	168	3,065
2	**Architecture inspection**	**85.00%**	10.00%	10.00%	2.50%	12.00%	14.93%
	Defects discovered	286	10	107	35	20	458
	Bad-fix injection	9	0	3	1	1	14
	Defects remaining	42	90	956	1,358	147	2,593
3	**Design inspection**	10.00%	14.00%	**87.00%**	7.00%	16.00%	37.30%
	Defects discovered	4	13	832	95	24	967
	Bad-fix injection	0	0	25	3	1	48
	Defects remaining	38	77	99	1,260	123	1,597
4	**Code inspection**	12.50%	15.00%	20.00%	85.00%	10.00%	70.10%
	Defects discovered	5	12	20	1,071	12	1,119
	Bad-fix injection	0	0	1	32	0	34
	Defects remaining	33	65	79	157	110	444
5	**Static Analysis**	2.00%	2.00%	7.00%	**87.00%**	3.00%	33.17%
	Defects discovered	1	1	6	136	3	147
	Bad-fix injection	0	0	0	4	0	4
	Defects remaining	32	64	73	16	107	292
6	**IV & V**	10.00%	12.00%	**23.00%**	7.00%	18.00%	16.45%
	Defects discovered	3	8	17	1	19	48
	Bad-fix injection	0	0	1	0	1	1
	Defects remaining	29	56	56	15	87	243
7	**SQA review**	10.00%	**17.00%**	**17.00%**	12.00%	12.50%	28.08%

	Defects discovered	3	10	9	2	11	35
	Bad-fix injection	0	0	0	0	0	2
	Defects remaining	26	46	46	13	76	206
	Pre-test DRE	**329**	**920**	**1,142**	**1,456**	**108**	**3,956**
	Pre-test DRE %	**92.73%**	**95.23%**	**96.12%**	**99.10%**	**58.79%**	**95.02%**
	Defects Remaining	**26**	**46**	**46**	**13**	**76**	**207**

TEST DEFECT REMOVAL ACTIVITIES

	Test Defect Removal Stages	**Architect.**	**Require.**	**Design**	**Code**	**Document**	**Total**
1	**Unit testing**	2.50%	4.00%	7.00%	**35.00%**	10.00%	8.69%
	Defects discovered	1	2	3	5	8	18
	Bad-fix injection	0	0	0	0	0	1
	Defects remaining	25	44	43	8	68	188
2	**Function testing**	7.50%	5.00%	22.00%	**37.50%**	10.00%	12.50%
	Defects discovered	2	2	9	3	7	23
	Bad-fix injection	0	0	0	0	0	1
	Defects remaining	23	42	33	5	61	164
3	**Regression testing**	2.00%	2.00%	5.00%	**33.00%**	7.50%	5.65%
	Defects discovered	0	1	2	2	5	9
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	23	41	31	3	56	154
4	**Integration testing**	6.00%	20.00%	22.00%	**33.00%**	15.00%	16.90%
	Defects discovered	1	8	7	1	8	26
	Bad-fix injection	0	0	0	0	0	1
	Defects remaining	21	33	24	2	48	127
5	**Performance testing**	14.00%	2.00%	**20.00%**	18.00%	2.50%	7.92%
	Defects discovered	3	1	5	0	1	10
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	18	32	19	2	46	117
6	**Security testing**	12.00%	15.00%	**23.00%**	8.00%	2.50%	10.87%
	Defects discovered	2	5	4	0	1	13
	Bad-fix injection	0	0	0	0	0	0

	Defects remaining	16	27	15	2	45	104
7	**Usability testing**	12.00%	17.00%	15.00%	5.00%	**48.00%**	29.35%
	Defects discovered	2	5	2	0	22	30
	Bad-fix injection	0	0	0	0	1	1
	Defects remaining	14	22	12	2	23	72
8	**System testing**	16.00%	12.00%	18.00%	12.00%	**34.00%**	20.85%
	Defects discovered	2	3	2	0	8	15
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	12	20	10	1	15	57
9	**Cloud testing**	10.00%	5.00%	13.00%	10.00%	**20.00%**	11.55%
	Defects discovered	1	1	1	0	3	7
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	10	19	9	1	12	51
10	**Independent testing**	12.00%	10.00%	11.00%	10.00%	**23.00%**	13.60%
	Defects discovered	1	2	1	0	3	7
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	9	17	8	1	9	44
11	**Field (Beta) testing**	14.00%	12.00%	14.00%	12.00%	**34.00%**	17.30%
	Defects discovered	1	2	1	0	3	8
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	8	15	7	1	6	36
12	**Acceptance testing**	13.00%	14.00%	15.00%	12.00%	**24.00%**	17.98%
	Defects discovered	1	2	1	0	2	6
	Bad-fix injection	0	0	0	0	0	0
	Defects remaining	7	13	6	1	3	30
	Test Defects Removed	**19**	**33**	**40**	**12**	**72**	**177**
	Testing Efficiency %	**73.96%**	**72.26%**	**87.63%**	**93.44%**	**95.45%**	**85.69%**
	Defects remaining	**7**	**13**	**6**	**1**	**3**	**30**
	Total Defects Removed	**348**	**953**	**1,183**	**1,468**	**181**	**4,133**
	Total Bad-fix injection	**10**	**29**	**35**	**44**	**5**	**124**
	Cumulative Removal %	**98.11%**	**98.68%**	**99.52%**	**99.94%**	**98.13%**	**99.27%**
	Remaining Defects	**7**	**13**	**6**	**1**	**3**	**30**
	High-severity Defects	**1**	**2**	**1**	**0**	**0**	**5**

Security Defects	**0**	**0**	**0**	**0**	**0**	**1**
Remaining Defects per Function Point	**0.0067**	**0.0128**	**0.0057**	**0.0009**	**0.0035**	**0.0302**
Remaining Defects per K Function Points	**6.72**	**12.80**	**5.70**	**0.87**	**3.45**	**30.23**
Remaining Defects per KLOC	**0.13**	**0.24**	**0.11**	**0.02**	**0.06**	**0.57**

Since the costs of finding and fixing bugs in software have been the largest single expense element for over 60 years, software quality and defect removal need the kind of data shown in table 3.

Defect Removal Efficiency Based on 90 Days after Release

It is obvious that measuring defect removal efficiency (DRE) based only on 30 days after release is insufficient to judge software quality:

Defects found before release	900	
Defects found in 30 days	5	99.45%
Defects found in 90 days	50	94.74%
Defects found in 360 days	75	92.31%

A 30 day interval after release will find very few defects since full usage may not even have begun due to installation and training. IBM selected a 90 day interval because that allowed normal usage patterns to unfold. Of course bugs continue to be found after 90 days, and also the software may be updated. A 90-day window is a good compromise for measuring defect removal efficiency of the original version before updates begin to accumulate.

A 30-day window may be sufficient for small projects < 250 function points. But anyone who has worked on large systems in the 10,000 to 100,000 function point size range knows that installation and training normally take about a month. Therefore full production may not even have started in the first 30 days.

Activity Based Benchmarks for Development

Today in 2017 software development is one of the most labor-intensive and expensive industrial activities in human history. Building large software applications costs more than the cost of a 50 story office building or the cost of an 80,000 ton cruise ship.

Given the fact that large software applications can employ more than 500 personnel in a total of more than 50 occupations, one might think that the industry would utilize fairly detailed activity-based benchmarks to explore the complexity of modern software development.

But unfortunately the majority of software benchmarks in 2016 are single values such as "work hours per function point," "function points per month," or "lines of code per month." This is not sufficient. Following are the kinds of activity-based benchmarks actually needed by the industry in order to understand the full economic picture of modern software development. Table 6 reflects a system of 10,000 function points and the Java programming language combined with an average team and iterative development:

Table 6: Example of Activity-based Benchmark

	Language	Java
	Function points	10,000
	Lines of code	533,333
	KLOC	533

	Development Activities	**Work Hours per FP**	**FP per month**	**Work Hours per KLOC**	**LOC per Month**
1	Business analysis	0.02	7,500.00	0.33	400,000
2	Risk analysis/sizing	0.00	35,000.00	0.07	1,866,666
3	Risk solution planning	0.01	15,000.00	0.17	800,000
4	Requirements	0.38	350.00	7.08	18,667
5	Requirement. Inspection	0.22	600.00	4.13	32,000
6	Prototyping	0.33	400.00	0.62	213,333
7	Architecture	0.05	2,500.00	0.99	133,333
8	Architecture. Inspection	0.04	3,000.00	0.83	160,000
9	Project plans/estimates	0.03	5,000.00	0.50	266,667
10	Initial Design	0.75	175.00	14.15	9,333
11	Detail Design	0.75	175.00	14.15	9,333
12	Design inspections	0.53	250.00	9.91	13,333
13	Coding	4.00	33.00	75.05	1,760
14	Code inspections	3.30	40.00	61.91	2,133
15	Reuse acquisition	0.01	10,000.00	0.25	533,333
16	Static analysis	0.02	7,500.00	0.33	400,000

17	COTS Package purchase	0.01	10,000.00	0.25	533,333
18	Open-source acquisition.	0.01	10,000.00	0.25	533,333
19	Code security audit.	0.04	3,500.00	0.71	186,667
20	Ind. Verification. & Validation (IV&V).	0.07	2,000.00	1.24	106,667
21	Configuration control.	0.04	3,500.00	0.71	186,667
22	Integration	0.04	3,500.00	0.71	186,667
23	User documentation	0.29	450.00	5.50	24,000
24	Unit testing	0.88	150.00	16.51	8,000
25	Function testing	0.75	175.00	14.15	9,333
26	Regression testing	0.53	250.00	9.91	13,333
27	Integration testing	0.44	300.00	8.26	16,000
28	Performance testing	0.33	400.00	6.19	21,333
29	Security testing	0.26	500.00	4.95	26,667
30	Usability testing	0.22	600.00	4.13	32,000
31	System testing	0.88	150.00	16.51	8,000
32	Cloud testing	0.13	1,000.00	2.48	53,333
33	Field (Beta) testing	0.18	750.00	3.30	40,000
34	Acceptance testing	0.05	2,500.00	0.99	133,333
35	Independent testing	0.07	2,000.00	1.24	106,667
36	Quality assurance	0.18	750.00	3.30	40,000
37	Installation/training	0.04	3,500.00	0.71	186,667
38	Project measurement	0.01	10,000.00	0.25	533,333
39	Project office	0.18	750.00	3.30	40,000
40	Project management	4.40	30.00	82.55	1,600
	Cumulative Results	**20.44**	**6.46**	**377.97**	**349**

Note that in real life non-code work such as requirements, architecture, and design are not measured using LOC metrics. But it is easy to retrofit LOC since the mathematics are not complicated. Incidentally the author's Software Risk Master (SRM) tool predicts all four values shown in table 6, and can also show story points, use case points, and in fact 23 different metrics.

The "cumulative results" show the most common benchmark form of single values. However single values are clearly inadequate to show the complexity of a full set of development activities.

Note that agile projects with multiple sprints would use a different set of activities. But to compare agile projects against other kinds of development methods the agile results are converted into a standard chart of accounts shown by table 4.

Note that there is no current equivalent to table 4 showing activity-based costs for SNAP metrics as of 2016. Indeed the IFPUG SNAP committee has not yet addressed the topic of activity-based costs.

Activity Based Benchmarks for Maintenance

The word "maintenance" is highly ambiguous and can encompass no fewer than 25 different kinds of work. In ordinary benchmarks "maintenance" usually refers to post-release defect repairs. However some companies and benchmarks also include enhancements. This is not a good idea since the funding for defect repairs and enhancements are from different sources, and often the work is done by different teams.

Table 7: Major Kinds of Work Performed Under the Generic Term "Maintenance"

1. Major Enhancements (new features of > 20 function points)
2. Minor Enhancements (new features of < 5 function points)
3. Maintenance (repairing defects for good will)
4. Warranty repairs (repairing defects under formal contract)
5. Customer support (responding to client phone calls or problem reports)
6. Error-prone module removal (eliminating very troublesome code segments)
7. Mandatory changes (required or statutory changes)
8. Complexity or structural analysis (charting control flow plus complexity metrics)
9. Code restructuring (reducing cyclomatic and essential complexity)
10. Optimization (increasing performance or throughput)
11. Migration (moving software from one platform to another)
12. Conversion (Changing the interface or file structure)
13. Reverse engineering (extracting latent design information from code)
14. Reengineering (transforming legacy application to modern forms)
15. Dead code removal (removing segments no longer utilized)
16. Dormant application elimination (archiving unused software)
17. Nationalization (modifying software for international use)
18. Mass updates such as Euro or Year 2000 Repairs

19. Refactoring, or reprogramming applications to improve clarity
20. Retirement (withdrawing an application from active service)
21. Field service (sending maintenance members to client locations)
22. Reporting bugs or defects to software vendors
23. Installing updates received from software vendors
24. Processing invalid defect reports
25. Processing duplicate defect reports

As with software development, function point metrics provide the most effective normalization metric for all forms of maintenance and enhancement work.

The author's Software Risk Master (SRM) tool predicts maintenance and enhancement for a three year period, and can also measure annual maintenance and enhancements. The entire set of metrics is among the most complex. However Table 7 illustrates a three-year pattern:

Table 7: Three-Year Maintenance, Enhancement, and Support Data

Enhancements (New Features)		**Year 1 2013**	**Year 2 2014**	**Year 3 2015**	**3-Year Totals**
Annual enhancement %	**8.00%**	200	216	233	**649**
Application Growth in FP	2,500	2,700	2,916	3,149	**3,149**
Application Growth in LOC	133,333	144,000	155,520	167,962	**167,962**
Cyclomatic complexity growth	10.67	10.70	10.74	10.78	**10.78**
Enhan. defects per FP	**0.01**	**0.00**	**0.00**	**0.00**	**0.00**
Enhan. defects delivered	**21**	**1**	**1**	**1**	**23**
Enhancement Team Staff	0	2.02	2.21	2.41	**2.22**
Enhancement (months)	0	24.29	26.51	28.94	**79.75**
Enhancement (hours)	0	3,206.48	3,499.84	3,820.47	**10,526.78**
Enhancement Team Costs	0	$273,279	$298,282	$325,608	**$897,169**
Function points per month		8.23	8.15	8.06	**8.14**
Work hours per function point		16.03	16.20	16.38	**16.21**
Enhancement $ per FP		$1,366.40	$1,380.93	$1,395.78	**$1,381.79**

Maintenance (Defect Repairs)		**Year 1 2013**	**Year 2 2014**	**Year 3 2015**	**3-Year Totals**
Number of maintenance sites	**1**	1	1	1	**1**
Clients served per site	74	94	118	149	**149**

Number of initial client sites	**3**	4	5	6	**6**
Annual rate of increase	**15.00%**	22.51%	22.51%	22.51%	**20.63%**
Number of initial clients	**100**	128	163	207	**207**
Annual rate of increase	**20.00%**	27.51%	27.51%	27.51%	**25.63%**
Client sites added	0	1	1	1	**3**
Client sites lost	0	0	0	0	**0**
Net change	0	1	1	1	**3**
Year end client sites	0	4	5	6	**6**
Clients added	0	28	36	46	**110**
Clients lost	0	-1	-1	-1	**-3**
Net change	0	28	35	45	**107**
Year end clients	0	128	163	207	207

Customer Defect/Help Requests		**Year 1 2013**	**Year 2 2014**	**Year 3 2015**	**3-Year Totals**
Customer satisfaction	**0**	**95.34%**	**99.42%**	**100.16%**	**98.31%**
Customer help requests	0	67	62	60	**189**
Customer complaints	0	24	18	15	**56**
Enhancement bug reports	**0**	**1**	**1**	**1**	**2**
Original bug reports	**0**	**8**	**5**	**3**	**16**
High severity bug reports	**0**	**1**	**1**	**0**	**2**
Security flaws	**0**	**1**	**0**	**0**	**0**
Bad fixes: bugs in repairs	**0**	**0**	**0**	**0**	**0**
Duplicate bug reports	0	8	7	6	**22**
Invalid bug reports	0	2	1	1	**4**
Abeyant defects	0	0	0	0	**0**
Total Incidents	**0**	**112**	**96**	**86**	**293**
Complaints per FP	0	0.01	0.01	0.01	**0.02**
Bug reports per FP	**0**	**0.00**	**0.00**	**0.00**	**0.01**
High severity bugs per FP	**0**	**0.00**	**0.00**	**0.00**	**0.00**
Incidents per FP	0	0.04	0.04	0.03	**0.12**

Maintenance and Support Staff	**Year 1 2013**	**Year 2 2014**	**Year 3 2015**	**3-Year Totals**

Customer support staff	0	0.31	0.33	0.38	**0.34**
Customer support (months)	0	3.72	4.01	4.56	**12.29**
Customer support (hours)	0	490.80	529.37	601.88	**1,622.05**
Customer support costs	0	$17,568	$18,949	$21,545	**$58,062**
Customer support $ per FP	0	$6.51	$6.50	$6.84	**$6.62**
Maintenance staff	0	1.83	1.80	1.77	**1.80**
Maintenance effort (months)	0	21.97	21.56	21.29	**64.82**
Maintenance effort (hours)	0	2,899.78	2,846.43	2,810.38	**8,556.59**
Maintenance (tech. debt)	0	$247,140	$242,593	$239,521	**$729,255**
Maintenance $ per FP	0	$91.53	$83.19	$76.06	**$83.59**
Management staff	0	0.22	0.22	0.22	**0.22**
Management effort (months)	0	2.69	2.66	2.67	**8.02**
Management effort (hours)	0	354.92	351.56	352.39	1,058.87
Management costs	0	$30,249	$29,963	$30,033	**$90,245**
Management $ per FP	0	$11.20	$10.28	$9.54	**$10.34**
TOTAL MAINTENANCE STAFF	**0**	**2.36**	**2.35**	**2.38**	**2.36**
TOTAL EFFORT (MONTHS)	**0**	**28.37**	**28.24**	**28.52**	**85.13**
TOTAL EFFORT (HOURS)	**0**	**3,745.50**	**3,727.36**	**3,764.66**	**11,237.51**
TOTAL MAINTENANCE $	**0**	**$294,957**	**$291,505**	**$291,099**	**$877,561**
Maintenance $ per FP	0	$117.98	$116.60	$116.44	**$117.01**
Maintenance hours per FP	0	1.39	1.28	1.20	**1.29**
Maintenance$ per defect	0	$32,865	$50,957	$82,650	**$55,490.43**
Maintenance $ per KLOC	0	$2,212	$2,186	$2,183	**$6,582**
Maintenance $ per incident	0	$2,637.01	$3,049.51	$3,375.50	**$3,020.67**
Incidents per support staff	0	360.99	286.03	226.96	**873.98**
Bug reports per staff member	0	11.57	8.52	6.42	**26.51**
Incidents per staff month	0	30.08	23.84	18.91	**24.28**
Bug reports per staff month	0	0.96	0.71	0.54	**0.74**

(MAINTENANCE + ENHANCMENT)

		Year 1 2013	**Year 2 2014**	**Year 3 2015**	**3-Year Totals**
Enhancement staff	0	2.02	2.21	2.41	2.22
Maintenance staff	0	2.36	2.35	2.38	2.36
Total staff	0	4.39	4.56	4.79	4.58

Enhancement effort (months)	0	24.29	26.51	28.94	79.75
Maintenance effort (months)	0	28.37	28.24	28.52	85.13
Total effort (months)	0	52.67	54.75	57.46	164.88
Total effort (hours)	0	6,951.97	7,227.19	7,585.12	21,764.29
Enhancement Effort %	0	46.12%	48.43%	50.37%	48.37%
Maintenance Effort %	0	53.88%	51.57%	49.63%	51.63%
Total Effort %	0	100.00%	100.00%	100.00%	100.00%
Enhancement cost	0	$273,279	$298,282	$325,608	$897,169
Maintenance cost	0	$294,957	$291,505	$291,099	$877,561
Total cost	0	$568,237	$589,786	$616,707	$1,774,730
Enhancement cost %	0	48.09%	50.57%	52.80%	50.55%
Maintenance cost %	0	51.91%	49.43%	47.20%	49.45%
Total Cost	0	100.00%	100.00%	100.00%	100.00%
Maintenance + Enhancement $ per FP		$210.46	$202.26	$195.82	$202.85
Maintenance + Enhancement hours per FP		2.57	2.48	2.41	2.49

The mathematical algorithms for predicting maintenance and enhancements can work for 10 year periods, but there is little value in going past three years since business changes or changes in government laws and mandates degrade long-range predictions.

Cost of Quality (COQ) for Quality Economics

The cost of quality (COQ) metric is roughly the same age as the software industry, having originated in 1956 by Edward Feigenbaum. It was later expanded by Joseph Juran and then made very famous by Phil Crosby in his seminal book "Quality is Free."

Quality was also dealt with fictionally in Robert M. Pirsig's famous book Zen and the Art of Motorcycle Maintenance. This book has become one of the best-selling books ever published and has been translated into many natural languages. It has sold over 5,000,000 copies. (By interesting coincidence Pirsig's regular work was as a software technical writer.)

Because COQ originated for manufacturing rather than for software, it needs to be modified slightly to be effective in a software context.

The original concepts of COQ include:

- Prevention costs
- Appraisal costs
- Internal failure costs
- External failure costs

- Total costs

For software a slightly modified set of topics for COQ include:

- Defect prevention costs (JAD, QFD, Kaizan, prototypes, etc.)
- Pre-Test defect removal costs (inspections, static analysis, pair programming, etc.)
- Test defect removal costs (unit, function, regression, performance, system, etc.)
- Post-release defect repairs costs (direct costs of defect repairs)
- Warranty and damage costs due to poor quality (fines, litigation, indirect costs)

Using round numbers and even values to simplify the concepts, the COQ results for a 20,000 function point application with average quality and Java might be:

Defect prevention	$1,500,000
Pre-test defect removal	$3,000,000
Test defect removal	$11,000,000
Post release repairs	$5,500,000
Damages and warranty costs	$3,000,000
Total Cost of Quality (COQ)	$24,000,000
COQ per function point	$1,200
COQ per KLOC	$24,000
COQ per SNAP point	Unknown as of 2016

If technical debt were included, but it not, the technical debt costs would probably be an additional $2,500,000. Among the issues with technical debt is that it focuses attention on a small subset of quality economic topics and of course does not deal with pre-release quality at all.

Total Cost of Ownership (TCO) for Software Economic Understanding

Because total cost of ownership cannot be measured or known until at least three years after release, it is seldom included in standard development benchmarks. The literature of TCO is sparse and there is very little reliable information. This is unfortunate because software TCO is much larger than the TCO of normal manufactured projects. This is due in part to poor quality control and in part to the continuous stream of enhancements which average about 8% per calendar year after the initial release, and sometimes runs for periods of more than 30 calendar years.

Another issue with TCO is that since applications continue to grow, after several years the size will have increased so much that the data needs to be renormalized with the current size. Table 5 illustrates a typical TCO estimate for an application that was 2,500 function points at delivery but grew to more than 3,000 function points after a three-year period:

Table 8: Example of Total Cost of Ownership (TCO) Estimates

	Staffing	Effort	Costs	$ per FP at release	% of TCO
Development	7.48	260.95	$3,914,201	$1,565.68	46.17%
Enhancement	2.22	79.75	$897,169	$358.87	10.58%
Maintenance	2.36	85.13	$877,561	$351.02	10.35%
Support	0.34	12.29	$58,062	$23.22	0.68%
User costs	4.20	196.69	$2,722,773	$1,089.11	32.12%
Additional costs			$7,500	$3.00	0.09%
Total TCO	**16.60**	**634.81**	**$8,477,266**	**$3,390.91**	**100.00%**

Function points at release	2,500
Function points after 3 years	3,149
Lines of code after 3 years	167,936
KLOC after 3 years	167.94
TCO function points/staff month	**4.96**
TCO work hours per function point	**26.61**
TCO cost per function point	**$2,692**
TCO cost per KLOC	**$50,479**

Note that as of 2017 there is no current data on TCO cost per SNAP point, nor even on a method for integrating SNAP into TCO calculations due to the fact that SNAP has not yet been applied to maintenance, enhancements, and user costs.

Note that the TCO costs include normal development, enhancement, maintenance, and customer support but also user costs. For internal project users participate in requirements, reviews, inspections, and other tasks so their costs and contributions should be shown as part of total cost of ownership (TCO).

Note that customer support costs are low because this particular application had only 100 users at delivery. Eventually users grew to more than 200 but initial defects declined so number of customer support personnel was only one person part time. Had this been a high-volume commercial application with 500,000 users that grew to over 1,000,000 users customer support would have included dozens of support personnel and grown constantly.

Note that for internal IT and web projects, operational costs can also be included in total costs of ownership. However operational costs are not relevant as TCO metrics for software that is run externally by external clients, such as software for automotive controls, avionics packages, medical devices such as cochlear implants, and commercial software sold or leased by companies such as Apple, Microsoft, IBM, and hundreds of others. It is also not a part of most open-source TCO studies.

Because applications grow at about 8% per year after release, the author suggests renormalizing application size at the end of every calendar year or every fiscal year. Table 8 shows a total growth pattern for 10 years. It is obvious that renormalization needs to occur fairly often due to the fact that all software applications grow over time as shown by table 8:

Table 8: SRM Multi-Year Sizing Example

Nominal application size in IFPUG function points	**10,000**
SNAP points	**1,389**
Language	**C**
Language level	**2.50**
Logical code statements	**1,280,000**

		Function Points	SNAP Points	Logical Code	
1	Size at end of requirements	10,000	1,389	1,280,000	
2	Size of requirement creep	2,000	278	256,000	
3	Size of planned delivery	12,000	1,667	1,536,000	
4	Size of deferred features	-4,800	(667)	(614,400)	
5	Size of actual delivery	7,200	1,000	921,600	
6	Year 1 usage	12,000	1,667	1,536,000	Kicker

7	Year 2 usage	13,000	1,806	1,664,000	
8	Year 3 usage	14,000	1,945	1,792,000	
9	Year 4 usage	17,000	2,361	2,176,000	Kicker
10	Year 5 usage	18,000	2,500	2,304,000	
11	Year 6 usage	19,000	2,639	2,432,000	
12	Year 7 usage	20,000	2,778	2,560,000	
13	Year 8 usage	23,000	3,195	2,944,000	Kicker
14	Year 9 usage	24,000	3,334	3,072,000	
15	Year 10 usage	25,000	3,473	3,200,000	

Kicker = Extra features added to defeat competitors.

Note: Simplified example with whole numbers for clarity.

Note: Deferred features usually due to schedule deadlines.

During development applications grow due to requirements creep at rates that range from below 1% per calendar month to more than 10% per calendar month. After release applications grow at rates that range from below 5% per year to more than 15% per year. Note that for commercial software "mid-life kickers" tend to occur about every four years. These are rich collections of new features intended to enhance competiveness.

Needs for Future Metrics

There is little research in the future metrics needs for the software industry. Neither universities nor corporations have devoted funds or effort into evaluating the accuracy of current metrics or creating important future metrics.

Some obvious needs for future metrics include:

1. Since companies own more data than software, there is an urgent need for a "*data point*" metric based on the logic of function point metrics. Currently neither data quality nor the costs of data acquisition can be estimated or measured due to the lack of a size metric for data.
2. Since many applications such as embedded software operate in specific devices, there is a need for a "*hardware function point*" metric based on the logic of function points.
3. Since web sites are now universal, there is a need for a "*web site point*" metric based on the logic of function points. This would measure web site contents.

4. Since risks are increasing for software projects, there is a need for a "*risk point*" metric based on the logic of function points.
5. Since cyber attacks are increasing in number and severity, there is a need for a "*security point*" metric based on the logic of function points.
6. Since software value includes both tangible financial value and also intangible value, there is a need for a "*value point*" metric based on the logic of function points.
7. Since software now has millions of human users in every country, there is a need for a "*software usage point*" metric based on the logic of function points.

The goal would be to generate integrated estimates.

Every major university and every major corporation should devote some funds and effort to the related topics of metrics validation and metrics expansion. It is professionally embarrassing for one of the largest industries in human history to have the least accurate and most ambiguous metrics of any industry for measuring the critical topics of productivity and quality.

Table 9 shows a hypothetical table of what integrated data might look like from a suite of related metrics that include software function points, hardware function points, data points, risk points, security points, and value points:

Table 9: Multi-Metric Economic

Development Metrics	**Number**	**Cost**	**Total**
Function points	1,000	$1,000	$1,000,000
Data points	1,500	$500	$750,000
Hardware function points	750	$2,500	$1,875,000
Subtotal	3,250	$1,115	$3,625,000
Annual Maintenance metrics			
Enhancements (micro function points)	150	$750	$112,500
Defects (micro function points)	750	$500	$375,000
Service points	5,000	$125	$625,000
Data maintenance	125	$250	$31,250
Hardware maintenance	200	$750	$150,000

Annual Subtotal	6,225	$179	$1,112,500
TOTAL COST OF OWNERSHIP (TCO)			
(Development + 5 years of usage)			
Development	3,250	$1,115	$3,625,000
Maintenance, enhancement, service	29,500	$189	$5,562,500
Data maintenance	625	$250	$156,250
Hardware maintenance	1,000	$750	$750,000
Application Total TCO	34,375	$294	$10,093,750
Risk and Value Metrics			
Risk points	2,000	$1,250	$2,500,000
Security points	1,000	$2,000	$2,000,000
Subtotal	3,000	$3,250	$4,500,000
Value points	45,000	$2,000	$90,000,000
NET VALUE	10,625	$7,521	$79,906,250
RETURN ON INVESTMENT (ROI)			$8.92

Note that as of 2017 the SNAP metric is not yet fully integrated into total software economic analysis.

Summary and Conclusions

Although function point metrics have solved many technical problems of software measurement, the current state of software metrics and measurement practices in 2017 is a professional embarrassment. Hundreds of companies in the software industry continue to use metrics proven mathematically to be invalid and which violate standard economic assumptions such as LOC and cost per defect.

Most universities do not carry out research studies on metrics validity but merely teach common metrics whether they work or not.

Until the software industry has a workable set of productivity and quality metrics that are standardized and widely used, progress will resemble a drunkard's walk. There are dozens of important topics that the software industry should know, but does not have effective data on circa 2017. Following are 21 samples where solid data would be valuable to the software industry:

Table 10: Twenty One Problems that Lack Effective Metrics and Data Circa 2017

1. How does agile quality and productivity compare to other methods?
2. Does agile work well for projects > 10,000 function points?
3. How effective is pair programming compared to inspections and static analysis?
4. Do ISO/IEC quality standards have any tangible results in lowering defect levels?
5. How effective is the new SEMAT method of software engineering?
6. What are best productivity rates for 100, 1000, 10,000, and 100,000 function points?
7. What are best quality results for 100, 1000, 10,000, and 100,000 function points?
8. What are the best quality results for CMMI levels 1, 2, 3, 4, and 5 for large systems?
9. What industries have the best software quality results?
10. What countries have the best software quality results?
11. How expensive are requirements and design compared to programming?
12. Do paper documents cost more than source code for defense software?
13. What is the optimal team size and composition for different kinds of software?
14. How does data quality compare to software quality?
15. How many delivered high-severity defects might indicate professional malpractice?
16. How often should software size be renormalized because of continuous growth?
17. How expensive is software governance?
18. What are the measured impacts of software reuse on productivity and quality?
19. What are the measured impacts of unpaid overtime on productivity and schedules?
20. What are the measured impacts of adding people to late software projects?
21. How does SNAP work for COQ, TCO, and activity-based costs?

These 21 issues are only the tip of the iceberg and dozens of other important topics are in urgent need of accurate predictions and accurate measurements. The software industry needs an effective suite of accurate and reliable metrics that can be used to predict and measure economic

productivity and application quality. Until we have such a suite of effective metrics, software engineering should not be considered to be a true profession.

Appendix A: Problems with Cost per Defect Metrics

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 2015:

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 three hidden problems with cost per defect that are usually not discussed in the software literature:

1. Cost per defect penalizes quality and is always cheapest where the greatest numbers of bugs are found.

2. 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.

3. 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.

The cost per defect metric has very serious shortcomings for economic studies of software quality. It penalizes high quality and ignores the major values of shorter schedules, lower development costs, lower maintenance costs, and lower warranty costs. In general cost per defect causes more harm than value as a software metric. Let us consider the cost per defect problem areas using examples that illustrate the main points.

Why Cost per Defect Penalizes Quality

The well-known and widely cited "cost per defect" measure unfortunately violates the canons of standard economics. Although this metric is often used to make quality economic claims, its main failing is that it penalizes quality and achieves the best results for the buggiest applications!

Furthermore, when zero-defect applications are reached there are still substantial appraisal and testing activities that need to be accounted for. Obviously the "cost per defect" metric is useless for zero-defect applications.

As with KLOC metrics discussed in Appendix B, the main source of error is that of ignoring fixed costs. Three examples will illustrate how "cost per defect" behaves as quality improves.

In all three cases, A, B, and C, we can assume that test personnel work 40 hours per week and are compensated at a rate of $2,500 per week or $75.75 per hour using fully burdened costs. Assume that all three software features that are being tested are 100 function points in size and 5000 lines of code in size (5 KLOC).

Case A: Poor Quality

Assume that a tester spent 15 hours writing test cases, 10 hours running them, and 15 hours fixing 10 bugs. The total hours spent was 40 and the total cost was $2,500. Since 10 bugs were found, the cost per defect was $250. The cost per function point for the week of testing would be $25.00. The cost per KLOC for the week of testing would be $500.

Case B: Good Quality

In this second case assume that a tester spent 15 hours writing test cases, 10 hours running them, and 5 hours fixing one bug, which was the only bug discovered.

However since no other assignments were waiting and the tester worked a full week 40 hours were charged to the project. The total cost for the week was still $2,500 so the cost per defect has jumped to $2,500.

If the 10 hours of slack time are backed out, leaving 30 hours for actual testing and bug repairs, the cost per defect would be $2,273.50 for the single bug. This is equal to $22.74 per function point or $454.70 per KLOC.

As quality improves, "cost per defect" rises sharply. The reason for this is that writing test cases and running them act like fixed costs. It is a well-known law of manufacturing economics that:

"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 11 shows the approximate costs for the three cost elements of preparation, execution, and repair for the test cycles just cited using the same rate of $:75.75 per hour for all activities:

Table 11: 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 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 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 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.

Let us now consider cost per function point as an alternative metric for measuring the costs of defect removal. With the slack removed the cost per function point would be $18.75. As can easily be seen cost per defect goes up as quality improves, thus violating the assumptions of standard economic measures.

However, as can also be seen, testing cost per function point declines as quality improves. This matches the assumptions of standard economics. The 10 hours of slack time illustrate another issue: when quality improves defects can decline faster than personnel can be reassigned.

Case C: Zero Defects

In this third case assume that a tester spent 15 hours writing test cases and 10 hours running them. No bugs or defects were discovered.

Because no defects were found, the "cost per defect" metric cannot be used at all. But 25 hours of actual effort were expended writing and running test cases. If the tester had no other assignments, he or she would still have worked a 40 hour week and the costs would have been $2,500.

If the 15 hours of slack time are backed out, leaving 25 hours for actual testing, the costs would have been $1,893.75. With slack time removed, the cost per function point would be $18.38. As can be seen again, testing cost per function point declines as quality improves. Here too, the decline in cost per function point matches the assumptions of standard economics.

Time and motion studies of defect repairs do not support the aphorism that "it costs 100 times as much to fix a bug after release as before." Bugs typically require between 15 minutes and 6 hours to repair regardless of where they are found.

(There are some bugs that are expensive and may takes several days to repair, or even longer. These are called "abeyant defects" by IBM. Abeyant defects are customer-reported defects

which the repair center cannot recreate, due to some special combination of hardware and software at the client site. Abeyant defects comprise less than 5% of customer-reported defects.)

Considering that cost per defect has been among the most widely used quality metrics for more than 50 years, the literature is surprisingly ambiguous about what activities go into "cost per defect." More than 75% of the articles and books that use cost per defect metrics do not state explicitly whether preparation and executions costs are included or excluded. In fact a majority of articles do not explain anything at all, but merely show numbers without discussing what activities are included.

Another major gap is that the literature is silent on variations in cost per defect by severity level. A study done by the author at IBM showed these variations in defect repair intervals associated with severity levels.

Table 12 shows the results of the study. Since these are customer-reported defects, "preparation and execution" would have been carried out by customers and the amounts were not reported to IBM. Peak effort for each severity level is highlighted in blue.

Table 12: Defect Repair Hours by Severity Levels for Field Defects

	Severity 1	Severity 2	Severity 3	Severity 4	Invalid	Average
> 40 hours	1.00%	3.00%	0.00%	0.00%	0.00%	0.80%
30 - 39 hours	3.00%	12.00%	1.00%	0.00%	1.00%	3.40%
20 - 29 hours	12.00%	20.00%	8.00%	0.00%	4.00%	8.80%
10 - 19 hours	22.00%	32.00%	10.00%	0.00%	12.00%	15.20%
1 - 9 hours	48.00%	22.00%	56.00%	40.00%	25.00%	38.20%
> 1 hour	14.00%	11.00%	25.00%	60.00%	58.00%	33.60%
TOTAL	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%

As can be seen, the overall average would be close to perhaps 5 hours, although the range is quite wide.

(As a matter of minor interest, the most troublesome bug found by the author during the time he was a professional programmer was a bug found during unit test, which took about 18 hours to analyze and repair. The software application where the bug occurred was an IBM 1401 program being ported to the larger IBM 1410 computer. The bug involved one instruction, which was valid on both the 1401 and 1410. However the two computers did not produce the same machine code. Thus the bug could not be found by examination of the source code itself, since that was

correct. The error could only be identified by examining the machine language generated for the two computers.)

In table 12, severity 1 defects mean that the software has stopped working. Severity 2 means that major features are disabled. Severity 3 refers to minor defects. Severity 4 defects are cosmetic in nature and do not affect operations. Invalid defects are hardware problems or customer errors inadvertently reported as software defects. A surprisingly large amount of time and effort goes into dealing with invalid defects although this topic is seldom discussed in the quality literature.

Yet another gap in the "cost per defect" literature is that of defect by origin. Following in Table 13 are typical results by defect origin points for 20 common defect types:

Table 13: Defect Repairs by Defect Origins

	Defect Origins	Find Hours	Repair Hours	Total Hours
1	**Security defects**	**11.00**	**24.00**	**35.00**
2	Errors of omission	8.00	24.00	32.00
3	Hardware errors	3.50	28.00	31.50
4	Abeyant defects	5.00	23.00	28.00
5	Data errors	1.00	26.00	27.00
6	Architecture defects	6.00	18.00	24.00
7	Toxic requirements	2.00	20.00	22.00
8	Requirements defects	5.00	16.50	21.50
9	Supply chain defects	6.00	11.00	17.00
10	Design defects	4.50	12.00	16.50
11	Structural defects	2.00	13.00	15.00
12	Performance defects	3.50	10.00	13.50
13	Bad test cases	5.00	7.50	12.50
14	Bad fix defects	3.00	9.00	12.00
15	Poor test coverage	4.50	2.00	6.50
16	Invalid defects	3.00	3.00	6.00
17	**Code defects**	**1.00**	**4.00**	**5.00**
18	Document defects	1.00	3.00	4.00
19	User errors	0.40	2.00	2.40
20	Duplicate defects	0.25	1.00	1.25
	Average	**3.78**	**12.85**	**16.63**

Table 13 shows "find hours" separately from "repair hours." The "find" tasks involve analysis of bug symptoms and the hardware/software combinations in use when the bug occurred. The "repair" tasks as the name implies are those of fixing the bug once it has been identified, plus regression testing to ensure the repair is not a "bad fix."

As can be seen, errors of omission, hardware errors, and data errors are the most expensive. Note also that errors caused by bad test cases and by "bad fixes" or secondary bugs in bug repairs themselves are more expensive than original code bugs. Note that even user errors and invalid defects require time for analysis and notifying clients of the situation.

The term "abeyant defects" originated in IBM circa 1965. It refers to defects that only occur for one client or one unique configuration of hardware and software. They are very hard to analyze and to fix.

Using Function Point Metrics for Defect Removal Economics

Because of the fixed or inelastic costs associated with defect removal operations, cost per defect always increases as numbers of defects decline. Because more defects are found at the beginning of a testing cycle than after release, this explains why cost per defect always goes up later in the cycle.

An alternate way of showing the economics of defect removal is to switch from "cost per defect" and use "defect removal cost per function point". Table 14 uses the same basic information as Table 11, but expresses all costs in terms of cost per function point:

Table 14 Cost per Function Point for Six Forms of Testing
(Assumes $75.75 per staff hour for costs)
(Assumes 100 function points in the application)

	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

The advantage of defect removal cost per function point over cost per defect is that it actually matches the assumptions of standard economics. In other words, as quality improves and defect volumes decline, cost per function point tracks these benefits and also declines. High quality is shown to be cheaper than poor quality, while with cost per defect high quality is incorrectly shown as being more expensive.

However, quality has more benefits to software applications than just those associated with defect removal activities. The most significant benefit of high quality is that it leads to shorter development schedules and cheaper overall costs for both development and maintenance. The total savings from high quality are much greater than the improvements in defect removal expenses.

Let us consider the value of high quality for a large system in the 10,000 function point size range.

The Value of Quality for Large Applications of 10,000 Function Points

When software applications reach 10,000 function points in size, they are very significant systems that require close attention to quality control, change control, and corporate governance. In fact without careful quality and change control, the odds of failure or cancellation top 35% for this size range.

Note that as application size increases, defect potentials increase rapidly and defect removal efficiency levels decline, even with sophisticated quality control steps in place. This is due to the exponential increase in the volume of paperwork for requirements and design, which often leads to partial inspections rather than 100% inspections. For large systems, test coverage declines and the number of test cases mounts rapidly but cannot usually keep pace with complexity.

Table 15: Quality Value for 10,000 Function Point Applications
(Note: 10,000 function points = 1,250,000 C statements)

	Average Quality	Excellent Quality	Difference
Defects per Function Point	6.00	3.50	-2.50
Defect Potential	60,000	35,000	-25,000
Defect Removal Efficiency	84.00%	96.00%	12.00%
Defects Removed	50,400	33,600	-16,800
Defects Delivered	9,600	1,400	-8,200
Cost per Defect	**$341**	**$417**	**$76**

Pre-Release			
Cost per Defect	**$833**	**$1,061**	**$227**
Post Release			
Development Schedule (Calendar Months)	40	28	-12
Development Staffing	67	67	0.00
Development Effort (Staff Months)	2,654	1,836	-818
Development Costs	$26,540,478	$18,361,525	-$8,178,953
Function Points per Staff Month	3.77	5.45	1.68
LOC per Staff Month	471	681	209.79
Maintenance Staff	17	17	0
Maintenance Effort (Staff Months)	800	117	-683.33
Maintenance Costs (Year 1)	$8,000,000	$1,166,667	-$6,833,333
TOTAL EFFORT (STAFF MONTHS)	3,454	1,953	-1501
TOTAL COST	$34,540,478	$19,528,191	-$15,012,287
TOTAL COST PER STAFF MEMBER	$414,486	$234,338	-$180,147
TOTAL COST PER FUNCTION POINT	**$3,454.05**	**$1,952.82**	**-$1,501.23**
TOTAL COST PER LOC	$27.63	$15.62	-$12.01
AVERAGE COST PER DEFECT	**$587**	**$739**	**$152**

The glaring problem of cost per defect is shown in table 15. Note that even though high quality reduced total costs by almost 50%, cost per defect is higher for the high-quality version than it is for the low-quality version! Note that cost per function point matches the true economic value of high quality, while "cost per defect" conceals the true economic value. Cost savings from better quality increase as application sizes increase. The general rule is that the larger the software application the more valuable quality becomes. The same principle is true for change control,

because the volume of creeping requirements goes up with application size.

Appendix B: Side by Side Comparisons of 79 Languages using LOC and Function Points

This appendix provides side-by-side comparisons of 79 programming languages using both function point metrics and lines of code metrics. Productivity is expressed using both hourly and monthly rates. The table assumes a constant value of 1000 function points for all 79 languages. However the number of lines of code varies widely based on the specific language.

Also held constant is the assumption for every language that the amount of non-code work for requirements, architecture, design, documentation, and management is an even 3000 hours.

As can be seen, Appendix B provides a mathematical proof that lines of code do not measure economic productivity. In Appendix B and in real life, economic productivity is defined as "*producing a specific quantity of goods for the lowest number of work hours.*"

Function points match this definition of economic productivity, but LOC metrics reverse true economic productivity and make the languages with the largest number of work hours seem more productive than the languages with the lowest number of work hours. Of course results for a single language will not have the problems shown in Appendix B.

In the following table "economic productivity" is shown in **green**, and is the "lowest number of work hours to deliver 1000 function points". Economic productivity is **NOT** "increasing the number of lines of code per month."

Although not shown in the table, it also includes a fixed value of 3,000 hours of non-code work for requirements, design, documents, management and the like. Thus "total work hours" in the table is the sum of code development + non-code effort. Since every language includes a constant value of 3,000 hours, this non-code effort is the "fixed cost" that drives up "cost per unit" when LOC declines. In real life the non-code work is a variable, but it simplifies the math and makes the essential point easier to see: LOC penalizes high-level languages.

Table 16: Side-by-Side Comparison of function points and lines of code metrics

	Languages	Size in KLOC	Total Work hours	Work hours per FP	FP per Month	Work Months	Work hours per KLOC	LOC per Month
1	Machine language	640.00	119,364	119.36	1.11	904.27	186.51	708
2	Basic Assembly	320.00	61,182	61.18	2.16	463.50	191.19	690
3	JCL	220.69	43,125	43.13	3.06	326.71	195.41	675
4	Macro Assembly	213.33	41,788	41.79	3.16	316.57	195.88	674
5	HTML	160.00	32,091	32.09	4.11	243.11	200.57	658
6	C	128.00	26,273	26.27	5.02	199.04	205.26	643
7	XML	128.00	26,273	26.27	5.02	199.04	205.26	643
8	Algol	106.67	22,394	22.39	5.89	169.65	209.94	629
9	Bliss	106.67	22,394	22.39	5.89	169.65	209.94	629
10	Chill	106.67	22,394	22.39	5.89	169.65	209.94	629
11	COBOL	106.67	22,394	22.39	5.89	169.65	209.94	629
12	Coral	106.67	22,394	22.39	5.89	169.65	209.94	629
13	Fortran	106.67	22,394	22.39	5.89	169.65	209.94	629
14	Jovial	106.67	22,394	22.39	5.89	169.65	209.94	629
15	GW Basic	98.46	20,902	20.90	6.32	158.35	212.29	622
16	Pascal	91.43	19,623	19.62	6.73	148.66	214.63	615
17	PL/S	91.43	19,623	19.62	6.73	148.66	214.63	615
18	ABAP	80.00	17,545	17.55	7.52	132.92	219.32	602
19	Modula	80.00	17,545	17.55	7.52	132.92	219.32	602
20	PL/I	80.00	17,545	17.55	7.52	132.92	219.32	602
21	ESPL/I	71.11	15,929	15.93	8.29	120.68	224.01	589
22	Javascript	71.11	15,929	15.93	8.29	120.68	224.01	589
23	Basic (interpreted)	64.00	14,636	14.64	9.02	110.88	228.69	577
24	Forth	64.00	14,636	14.64	9.02	110.88	228.60	577

25	haXe	64.00	**14,636**	**14.64**	**9.02**	110.88	**228.69**	**577**
26	Lisp	64.00	**14,636**	**14.64**	**9.02**	110.88	**228.69**	**577**
27	Prolog	64.00	**14,636**	**14.64**	**9.02**	110.88	**228.69**	**577**
28	SH (shell scripts)	64.00	**14,636**	**14.64**	**9.02**	110.88	**228.69**	**577**
29	Quick Basic	60.95	**14,082**	**14.08**	**9.37**	106.68	**231.04**	**571**
30	Zimbu	58.18	**13,579**	**13.58**	**9.72**	102.87	**233.38**	**566**
31	C++	53.33	**12,697**	**12.70**	**10.40**	96.19	**238.07**	**554**
32	Go	53.33	**12,697**	**12.70**	**10.40**	96.19	**238.07**	**554**
33	Java	53.33	**12,697**	**12.70**	**10.40**	96.19	**238.07**	**554**
34	PHP	53.33	**12,697**	**12.70**	**10.40**	96.19	**238.07**	**554**
35	Python	53.33	**12,697**	**12.70**	**10.40**	96.19	**238.07**	**554**
36	C#	51.20	**12,309**	**12.31**	**10.72**	93.25	**240.41**	**549**
37	X10	51.20	**12,309**	**12.31**	**10.72**	93.25	**240.41**	**549**
38	Ada 95	49.23	**11,951**	**11.95**	**11.05**	90.54	**242.76**	**544**
39	Ceylon	49.23	**11,951**	**11.95**	**11.05**	90.54	**242.76**	**544**
40	Fantom	49.23	**11,951**	**11.95**	**11.05**	90.54	**242.76**	**544**
41	Dart	47.41	**11,620**	**11.62**	**11.36**	88.03	**245.10**	**539**
42	RPG III	47.41	**11,620**	**11.62**	**11.36**	88.03	**245.10**	**539**
43	CICS	45.71	**11,312**	**11.31**	**11.67**	85.69	**247.44**	**533**
44	DTABL	45.71	**11,312**	**11.31**	**11.67**	85.69	**247.44**	**533**
45	F#	45.71	**11,312**	**11.31**	**11.67**	85.69	**247.44**	**533**
46	Ruby	45.71	**11,312**	**11.31**	**11.67**	85.69	**247.44**	**533**
47	Simula	45.71	**11,312**	**11.31**	**11.67**	85.69	**247.44**	**533**
48	Erlang	42.67	**10,758**	**10.76**	**12.27**	81.50	**252.13**	**524**
49	DB2	40.00	**10,273**	**10.27**	**12.85**	77.82	**256.82**	**514**
50	LiveScript	40.00	**10,273**	**10.27**	**12.85**	77.82	**256.82**	**514**
51	Oracle	40.00	**10,273**	**10.27**	**12.85**	77.82	**256.82**	**514**
52	Elixir	37.65	**9,845**	**9.84**	**13.41**	74.58	**261.51**	**505**

53	Haskell	37.65	**9,845**	**9.84**	**13.41**	74.58	**261.51**	**505**
54	Mixed Languages	37.65	**9,845**	**9.84**	**13.41**	74.58	**261.51**	**505**
55	Julia	35.56	**9,465**	**9.46**	**13.95**	71.70	**266.19**	**496**
56	M	35.56	**9,465**	**9.46**	**13.95**	71.70	**266.19**	**496**
57	OPA	35.56	**9,465**	**9.46**	**13.95**	71.70	**266.19**	**496**
58	Perl	35.56	**9,465**	**9.46**	**13.95**	71.70	**266.19**	**496**
59	APL	32.00	**8,818**	**8.82**	**14.97**	66.80	**275.57**	**479**
60	Delphi	29.09	**8,289**	**8.29**	**15.92**	62.80	**284.94**	**463**
61	Objective C	26.67	**7,848**	**7.85**	**16.82**	59.46	**294.32**	**448**
62	Visual Basic	26.67	**7,848**	**7.85**	**16.82**	59.46	**294.32**	**448**
63	ASP NET	24.62	**7,476**	**7.48**	**17.66**	56.63	**303.69**	**435**
64	Eiffel	22.86	**7,156**	**7.16**	**18.45**	54.21	**313.07**	**422**
65	Smalltalk	21.33	**6,879**	**6.88**	**19.19**	52.11	**322.44**	**409**
66	IBM ADF	20.00	**6,636**	**6.64**	**19.89**	50.28	**331.82**	**398**
67	MUMPS	18.82	**6,422**	**6.42**	**20.55**	48.65	**341.19**	**387**
68	Forte	17.78	**6,232**	**6.23**	**21.18**	47.21	**350.57**	**377**
69	APS	16.84	**6,062**	**6.06**	**21.77**	45.93	**359.94**	**367**
70	TELON	16.00	**5,909**	**5.91**	**22.34**	44.77	**369.32**	**357**
71	Mathematica9	12.80	**5,327**	**5.33**	**24.78**	40.36	**416.19**	**317**
72	TranscriptSQL	12.80	**5,327**	**5.33**	**24.78**	40.36	**416.19**	**317**
73	QBE	12.80	**5,327**	**5.33**	**24.78**	40.36	**416.19**	**317**
74	X	12.80	**5,327**	**5.33**	**24.78**	40.36	**416.19**	**317**
75	Mathematica10	9.14	**4,662**	**4.66**	**28.31**	35.32	**509.94**	**259**
76	BPM	7.11	**4,293**	**4.29**	**30.75**	32.52	**603.69**	**219**
77	Generators	7.11	**4,293**	**4.29**	**30.75**	32.52	**603.69**	**219**
78	Excel	6.40	**4,164**	**4.16**	**31.70**	31.54	**650.57**	**203**
79	IntegraNova	5.33	**3,970**	**3.97**	**33.25**	30.07	**744.32**	**177**

Average	**67.60**	**15,291**	**15.29**	**12.80**	**115.84**	**279.12**	**515**

It is obvious that in real life no one would produce 1000 function points in machine language, JCL, or some of the other languages in the table. The table is merely illustrative of the fact that while function points may be constant and non-code hours are fixed costs, coding effort is variable and proportional to the amount of source code produced.

In Table 16 the exact number of KLOC can vary language to language, from team to team, and company to company. But that is irrelevant to the basic mathematics of the case. There are three aspects to the math:

Point 1: When a manufacturing process includes a high proportion of fixed costs and there is a reduction in the units produced, the cost per unit will go up. This is true for all industries and all manufactured products without exception.

Point 2: When switching from a low-level programming language to a high-level programming language, the number of "units" produced will be reduced.

Point 3: The reduction in LOC metrics for high-level languages in the presence of the fixed costs for requirements and design will cause cost per LOC to go up and will also cause LOC per month to come down for high-level languages.

These three points are nothing more than the standard rules of manufacturing economics applied to software and programming languages.

The LOC metric originated in the 1950's when machine language and basic assembly were the only languages in use. In those early days coding was over 95% of the total effort so the fixed costs of non-code work barely mattered. It was only after high-level programming languages began to reduce coding effort and requirements and design became progressively larger components that the LOC problems occurred. Table 17 shows the coding and non-coding percentages by language with the caveat that the non-code work is artificially held constant at 3000 hours:

Table 17: Percentages of Coding and Non-Coding Tasks
(Percent of work hours for code and non-code)

	Languages	Non-code Percent	Code Percent
1	Machine language	2.51%	97.49%
2	Basic Assembly	4.90%	95.10%
3	JCL	6.96%	93.04%
4	Macro Assembly	7.18%	92.82%
5	HTML	9.35%	90.65%
6	C	11.42%	88.58%
7	XML	11.42%	88.58%
8	Algol	13.40%	86.60%
9	Bliss	13.40%	86.60%
10	Chill	13.40%	86.60%
11	COBOL	13.40%	86.60%
12	Coral	13.40%	86.60%
13	Fortran	13.40%	86.60%
14	Jovial	13.40%	86.60%
15	GW Basic	14.35%	85.65%
16	Pascal	15.29%	84.71%
17	PL/S	15.29%	84.71%
18	ABAP	17.10%	82.90%
19	Modula	17.10%	82.90%
20	PL/I	17.10%	82.90%
21	ESPL/I	18.83%	81.17%
22	Javascript	18.83%	81.17%
23	Basic (interpreted)	20.50%	79.50%
24	Forth	20.50%	79.50%
25	haXe	20.50%	79.50%
26	Lisp	20.50%	79.50%
27	Prolog	20.50%	79.50%
28	SH (shell scripts)	20.50%	79.50%
29	Quick Basic	21.30%	78.70%
30	Zimbu	22.09%	77.91%
31	C++	23.63%	76.37%
32	Go	23.63%	76.37%
33	Java	23.63%	76.37%
34	PHP	23.63%	76.37%
35	Python	23.63%	76.37%

36	C#	24.37%	75.63%
37	X10	24.37%	75.63%
38	Ada 95	25.10%	74.90%
39	Ceylon	25.10%	74.90%
40	Fantom	25.10%	74.90%
41	Dart	25.82%	74.18%
42	RPG III	25.82%	74.18%
43	CICS	26.52%	73.48%
44	DTABL	26.52%	73.48%
45	F#	26.52%	73.48%
46	Ruby	26.52%	73.48%
47	Simula	26.52%	73.48%
48	Erlang	27.89%	72.11%
49	DB2	29.20%	70.80%
50	LiveScript	29.20%	70.80%
51	Oracle	29.20%	70.80%
52	Elixir	30.47%	69.53%
53	Haskell	30.47%	69.53%
54	Mixed Languages	30.47%	69.53%
55	Julia	31.70%	68.30%
56	M	31.70%	68.30%
57	OPA	31.70%	68.30%
58	Perl	31.70%	68.30%
59	APL	34.02%	65.98%
60	Delphi	36.19%	63.81%
61	Objective C	38.22%	61.78%
62	Visual Basic	38.22%	61.78%
63	ASP NET	40.13%	59.87%
64	Eiffel	41.92%	58.08%
65	Smalltalk	43.61%	56.39%
66	IBM ADF	45.21%	54.79%
67	MUMPS	46.71%	53.29%
68	Forte	48.14%	51.86%
69	APS	49.49%	50.51%
70	TELON	50.77%	49.23%
71	Mathematica9	56.31%	43.69%
72	TranscriptSQL	56.31%	43.69%
73	QBE	56.31%	43.69%
74	X	56.31%	43.69%
75	Mathematica10	64.35%	35.65%
76	BPM	69.88%	30.12%
77	Generators	69.88%	30.12%
78	Excel	72.05%	27.95%

79	IntegraNova	75.57%	24.43%
	Average	**29.08%**	**70.92%**

As can easily be seen for very low-level languages the problems of LOC metrics are minor. But as language levels increase, a higher percentage of effort goes to non-code work while coding effort progressively gets smaller. Thus LOC metrics are invalid and hazardous for high-level languages.

It might be thought that omitting non-code effort and only showing coding may preserve the usefulness of LOC metrics, but this is not the case. Productivity is still producing deliverable for the lowest number of work hours or the lowest amount of effort.

Producing a feature in 500 lines of Objective-C at a rate of 500 LOC per month has better economic productivity than producing the same feature in 1000 lines of Java at a rate of 600 LOC per month.

Objective-C took 1 month or 149 work hours for the feature. Java took 1.66 months or 247 hours. Even though coding speed favors Java by a rate of 600 LOC per month to 500 LOC per month for Objective-C, economic productivity clearly belongs to Objective-C because of the reduced work effort.

Function points were specifically invented by IBM to measure economic productivity. Function point metrics stay constant no matter what programming language is used. Therefore function points are not troubled by the basic rule of manufacturing economics that when a process has fixed costs and the number of units goes down, cost per unit goes up. Function points are the same regardless of programming languages. Thus in today's world of 2014 function point metrics measure software economic productivity, but LOC metrics do not.

References and Readings

Books and monographs by Capers Jones.

New in 2017

Jones, Capers; A Guide to Selecting Software Measures and Metrics; CRC Press; April 2017.

Older books by Capers Jones

1 Jones, Capers; The Technical and Social History of Software Engineering; Addison Wesley 2014
2 Jones, Capers & Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley, 2012
3 Jones, Capers; Software Engineering Best Practices; 1st edition; McGraw Hill 2010
4 Jones, Capers: Applied Software Measurement; 3rd edition; McGraw Hill 2008
5 Jones, Capers: Estimating Software Costs, 2nd edition; McGraw Hill 2007
6 Jones, Capers: Software Assessments, Benchmarks, and Best Practices; Addison Wesley, 2000
7 Jones, Capers: Software Quality - Analysis and Guidelines for Success, International Thomson 1997
8 Jones, Capers; Patterns of Software Systems Failure and Success; International Thomson 1995
9 Jones, Capers; Assessment and Control of Software Risks; Prentice Hall 1993
10 Jones, Capers: Critical Problems in Software Measurement; IS Mgt Group 1993

Monographs by Capers Jones 2012-2017 available from Namcook Analytics LLC

1 Comparing Software Development Methodologies
2 Corporate Software Risk Reduction
3 Defenses Against Breach of Contract Litigation
4 Dynamic Visualization of Software Development
5 Evaluation of Common Software Metrics
6 Function Points as a Universal Software Metric
7 Hazards of "cost per defect" metrics
8 Hazards of "lines of code" metrics
9 Hazards of "technical debt" metrics
10 History of Software Estimation Tools
11 How Software Engineers Learn New Skills
12 Software Benchmark Technologies
13 Software Defect Origins and Removal Methods
14 Software Defect Removal Efficiency (DRE)
15 Software Project Management Tools

Books by other authors:

Abrain, Alain; Software Estimating Models; Wiley-IEEE Computer Society; 2015

Abrain, Alain; Software Metrics and Metrology; Wiley-IEEE Computer Society; 2010

Abrain, Alain; Software Maintenance Management: Evolution and Continuous Improvement; Wiley-IEEE Computer Society, 2008.

Albrecht, Allan; AD/M Productivity Measurement and Estimate Validation; IBM Corporation, Purchase, NY; May 1984.

Barrow, Dean, Nilson, Susan, and Timberlake, Dawn; Software Estimation Technology Report; Air Force Software Technology Support Center; Hill Air Force Base, Utah; 1993.

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

Brooks, Fred; The Mythical Man Month; Addison-Wesley, Reading, MA; 1995; 295 pages.

Bundschuh, Manfred and Dekkers, Carol; The IT Measurement Compendium; Springer-Verlag, Berlin; 2008; 643 pages.

Brown, Norm (Editor); The Program Manager's Guide to Software Acquisition Best Practices; Version 1.0; July 1995; U.S. Department of Defense, Washington, DC; 142 pages.

Chidamber, S.R. and Kemerer, C.F.: "A Metrics Suite for Object Oriented Design"; IEEE Transactions on Software Engineering; Vol. 20, 1994; pp. 476-493.

Chidamber, S.R., Darcy, D.P., and Kemerer, C.F.: "Managerial Use of Object Oriented Software Metrics"; Joseph M. Katz Graduate School of Business, University of Pittsburgh, Pittsburgh, PA; Working Paper # 750; November 1996; 26 pages.

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

Conte, S.D., Dunsmore, H.E., and Shen, V.Y.; Software Engineering Models and Metrics; The Benjamin Cummings Publishing Company, Menlo Park, CA; ISBN 0-8053-2162-4; 1986; 396 pages.

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

DeMarco, Tom and Lister, Tim; Peopleware; Dorset House Press, New York, NY; 1987; ISBN 0-932633-05-6; 188 pages.

DeMarco, Tom; Why Does Software Cost So Much?; Dorset House Press, New York, NY; ISBN 0-932633-34-X; 1995; 237 pages.

DeMarco, Tom; Deadline; Dorset House Press, New York, NY; 1997.

Department of the Air Force; Guidelines for Successful Acquisition and Management of Software Intensive Systems; Volumes 1 and 2; Software Technology Support Center, Hill Air Force Base, UT; 1994.

Dreger, Brian; Function Point Analysis; Prentice Hall, Englewood Cliffs, NJ; 1989; ISBN 0-13-332321-8; 185 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.

Galea, R.B.; The Boeing Company: 3D Function Point Extensions, V2.0, Release 1.0; Boeing Information Support Services, Seattle, WA; June 1995.

Galorath, Daniel D. and Evans, Michael W.; Software Sizing, Estimation, and Risk Management; Auerbach Publications, New York, 2006.

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

Garmus, David & Herron, David; Function Point Analysis; Addison Wesley Longman, Boston, MA; 1996.

Garmus, David; Accurate Estimation; *Software Development*; July 1996; pp 57-65.

Grady, Robert B.; Practical Software Metrics for Project Management and Process Improvement; Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-720384-5; 1992; 270 pages.

Grady, Robert B. & Caswell, Deborah L.; Software Metrics: Establishing a Company-Wide Program; Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-821844-7; 1987; 288 pages.

Gulledge, Thomas R., Hutzler, William P.; and Lovelace, Joan S.(Editors); Cost Estimating and Analysis - Balancing Technology with Declining Budgets; Springer-Verlag; New York; ISBN 0-387-97838-0; 1992; 297 pages.

Harris, Michael D.S., Herron, David, and Iwanacki, Stasia; The Business Value of IT; CRC Press, Auerbach Publications; 2009.

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

Howard, Alan (Ed.); Software Metrics and Project Management Tools; Applied Computer Research (ACR; Phoenix, AZ; 1997; 30 pages.

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

Humphrey, Watts; Personal Software Process; Addison Wesley Longman, Reading, MA; 1997.

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.

Kemerer, Chris F.; "An Empirical Validation of Software Cost Estimation Models; Communications of the ACM; 30; May 1987; pp. 416-429.

Kemerer, C.F.; "Reliability of Function Point Measurement - A Field Experiment"; Communications of the ACM; Vol. 36; pp 85-97; 1993.

Keys, Jessica; Software Engineering Productivity Handbook; McGraw Hill, New York, NY; ISBN 0-07-911366-4; 1993; 651 pages.

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.

Love, Tom; Object Lessons; SIGS Books, New York; ISBN 0-9627477 3-4; 1993; 266 pages.

Marciniak, John J. (Editor); Encyclopedia of Software Engineering; John Wiley & Sons, New York; 1994; ISBN 0-471-54002; in two volumes.

McCabe, Thomas J.; "A Complexity Measure"; IEEE Transactions on Software Engineering; December 1976; pp. 308-320.

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

Melton, Austin; Software Measurement; International Thomson Press, London, UK; ISBN 1-85032-7178-7; 1995.

Mertes, Karen R.; Calibration of the CHECKPOINT Model to the Space and Missile Systems Center (SMC) Software Database (SWDB); Thesis AFIT/GCA/LAS/96S-11, Air Force Institute of Technology (AFIT), Wright Patterson AFB, Ohio; September 1996; 119 pages.

Mills, Harlan; Software Productivity; Dorset House Press, New York, NY; ISBN 0-932633-10-2; 1988; 288 pages.

Muller, Monika & Abram, Alain (editors); Metrics in Software Evolution; R. Oldenbourg Vertag GmbH, Munich; ISBN 3-486-23589-3; 1995.

Multiple authors; Rethinking the Software Process; (CD-ROM); Miller Freeman, Lawrence, KS; 1996. (This is a new CD ROM book collection jointly produced by the book publisher, Prentice Hall, and the journal publisher, Miller Freeman. This CD ROM disk contains the full text and illustrations of five Prentice Hall books: Assessment and Control of Software Risks by Capers Jones; Controlling Software Projects by Tom DeMarco; Function Point Analysis by Brian Dreger; Measures for Excellence by Larry Putnam and Ware Myers; and Object-Oriented Software Metrics by Mark Lorenz and Jeff Kidd.)

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.

Paulk Mark et al; The Capability Maturity Model; Guidelines for Improving the Software Process; Addison Wesley, Reading, MA; ISBN 0-201-54664-7; 1995; 439 pages.

Perlis, Alan J., Sayward, Frederick G., and Shaw, Mary (Editors); Software Metrics; The MIT Press, Cambridge, MA; ISBN 0-262-16083-8; 1981; 404 pages.

Perry, William E.; Data Processing Budgets - How to Develop and Use Budgets Effectively; Prentice Hall, Englewood Cliffs, NJ; ISBN 0-13-196874-2; 1985; 224 pages.

Perry, William E.; Handbook of Diagnosing and Solving Computer Problems; TAB Books, Inc.; Blue Ridge Summit, PA; 1989; ISBN 0-8306-9233-9; 255 pages.

Pressman, Roger; Software Engineering - A Practitioner's Approach; McGraw Hill, New York, NY; 1982.

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

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

Reifer, Donald (editor); Software Management (4th edition); IEEE Press, Los Alamitos, CA; ISBN 0 8186-3342-6; 1993; 664 pages.

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

Royce, W.E.; Software Project Management: A Unified Framework; Addison Wesley, Reading, MA; 1999

Rubin, Howard; Software Benchmark Studies For 1997; Howard Rubin Associates, Pound Ridge, NY; 1997.

Shepperd, M.: "A Critique of Cyclomatic Complexity as a Software Metric"; Software Engineering Journal, Vol. 3, 1988; pp. 30-36.

Software Productivity Consortium; The Software Measurement Guidebook; International Thomson Computer Press; Boston, MA; ISBN 1-850-32195-7; 1995; 308 pages.

St-Pierre, Denis; Maya, Marcela; Abran, Alain, and Desharnais, Jean-Marc; Full Function Points: Function Point Extensions for Real-Time Software, Concepts and Definitions; University of Quebec. Software Engineering Laboratory in Applied Metrics (SELAM); TR 1997-03; March 1997; 18 pages.

Strassmann, Paul; The Squandered Computer; The Information Economics Press, New Canaan, CT; ISBN 0-9620413-1-9; 1997; 426 pages.

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.

Stutzke, Richard D.; Estimating Software Intensive Systems; Addison Wesley, Boston, MA; 2005.

Symons, Charles R.; Software Sizing and Estimating – Mk II FPA (Function Point Analysis); John Wiley & Sons, Chichester; ISBN 0 471-92985-9; 1991; 200 pages.

Thayer, Richard H. (editor); Software Engineering and Project Management; IEEE Press, Los Alamitos, CA; ISBN 0 8186-075107; 1988; 512 pages.

Umbaugh, Robert E. (Editor); Handbook of IS Management; (Fourth Edition); Auerbach Publications, Boston, MA; ISBN 0-7913-2159-2; 1995; 703 pages.

Whitmire, S.A.; "3-D Function Points: Scientific and Real-Time Extensions to Function Points"; Proceedings of the 1992 Pacific Northwest Software Quality Conference, June 1, 1992.

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

Zells, Lois; Managing Software Projects - Selecting and Using PC-Based Project Management Systems; QED Information Sciences, Wellesley, MA; ISBN 0-89435-275-X; 1990; 487 pages.

Zuse, Horst; Software Complexity - Measures and Methods; Walter de Gruyter, Berlin; 1990; ISBN 3-11-012226-X; 603 pages.

Zuse, Horst; A Framework of Software Measurement; Walter de Gruyter, Berlin; 1997.

	Software Benchmark Providers (listed in alphabetic order)	
1	4SUM Partners	www.4sumpartners.com
2	Bureau of Labor Statistics, Department of Commerce	www.bls.gov
3	Capers Jones (Namcook Analytics LLC)	www.namcook.com
4	CAST Software	www.castsoftware.com
5	Congressional Cyber Security Caucus	cybercaucus.langevin.house.gov
6	Construx	www.construx.com
7	COSMIC function points	www.cosmicon.com
8	Cyber Security and Information Systems	https://s2cpat.thecsiac.com/s2cpat/
9	David Consulting Group	www.davidconsultinggroup.com
10	Forrester Research	www.forrester.com
11	Galorath Incorporated	www.galorath.com
12	Gartner Group	www.gartner.com
13	German Computer Society	http://metrics.cs.uni-magdeburg.de/
14	Hoovers Guides to Business	www.hoovers.com
15	IDC	www.IDC.com
16	ISBSG Limited	www.isbsg.org
17	ITMPI	www.itmpi.org
18	Jerry Luftman (Stevens Institute)	http://howe.stevens.edu/index.php?id=14
19	Level 4 Ventures	www.level4ventures.com
20	Namcook Analytics LLC	www.namcook.com
21	Price Systems	www.pricesystems.com
22	Process Fusion	www.process-fusion.net
23	QuantiMetrics	www.quantimetrics.net
24	Quantitative Software Management (QSM)	www.qsm.com
25	Q/P Management Group	www.qpmg.com
26	RBCS, Inc.	www.rbcs-us.com
27	Reifer Consultants LLC	www.reifer.com
28	Howard Rubin	www.rubinworldwide.com
29	SANS Institute	www.sabs,org
30	Software Benchmarking Organization (SBO)	www.sw-benchmark.org
31	Software Engineering Institute (SEI)	www.sei.cmu.edu
32	Software Improvement Group (SIG)	www.sig,eu
33	Software Productivity Research	www.SPR.com
34	Standish Group	www.standishgroup.com
35	Strassmann, Paul	www.strassmann.com
36	System Verification Associates LLC	http://sysverif.com
37	Test Maturity Model Integrated	www.experimentus.com