ERRORS AND OMISSIONS IN SOFTWARE HISTORICAL DATA:

SEPARATING FACT FROM FICTION

Version 4.0

August 17, 2009

Abstract

In order to understand the economic issues of software development and maintenance, it is important to have accurate historical data. Unfortunately what is called "historical data" by the software industry is usually less than 50% complete and sometimes less than 20% complete. The most common omissions include unpaid overtime, managerial effort, and the work of part-time specialists such as quality assurance, technical writers, data-base administrators, function point specialists, and a number of other occupations.

The most common result from these gaps and omissions in historical data is to artificially inflate productivity, and make projects look faster and cheaper than they really are. This error, in turn, leads to optimistic cost estimates if the "historical data" is taken at face value and used without correction.

Software quality data also has significant gaps and errors. Many companies do not measure defects until after delivery of the software to customers. Other companies measure defects found in system test, but not in earlier tests or inspections. Comparatively few start defect measurements early during reviews and inspections.

In addition to these gaps in historical data, the software industry also uses metrics that violate standard economic principles and distort results in very significant ways. The two most common flawed metrics include "cost per defect" which penalizes quality, and "lines of code" which penalize high-level programming languages.

Capers Jones
President, Capers Jones & Associates LLC
CJonesiii@cs.com

TABLE OF CONTENTS

ERRORS AND OMISSIONS IN SOFTWARE HISTORICAL DATA

INTRODUCTION

As the developer of a family of software cost estimating tools, the author is often asked what seems to be a straight-forward question: "*How accurate are the estimates compared to historical data?*" The answer to this question is surprising. Usually the estimates are far more accurate than the historical data, because the "historical data" is incomplete and omits most of the actual costs and work effort that were accrued. In some cases "historical data" only captures 20% or less of the full amount of effort that was expended.

Thus when the outputs from an accurate software cost estimating tool are compared to what is called "historical data" the results tend to be alarming. The outputs from the estimating tool often indicate higher costs, more effort, and longer schedules than the historical data indicates. It is seldom realized that the difference is because of major gaps and omissions in the historical data itself, rather than because of errors in the estimates.

It is fair to ask if historical data is incomplete, how is it possible to know the true amounts and judge the quantity of missing data that was omitted?

In order to correct the gaps and omissions that are normal in cost tracking systems, it is necessary to interview the development team members and the project managers. During these interview sessions, the contents of the historical data collected for the project are compared to a complete work breakdown structure derived from similar projects. For each activity and task that occurs in the work breakdown structure, but which is missing from the historical data, the developers are asked whether or not the activity occurred. If it did occur, the developers are asked to reconstruct from memory or their informal records the number of hours that the missing activity accrued.

Problems with errors and "leakage" from software cost tracking systems are as old as the software industry itself. The first edition of the author's book Applied Software Measurement was published in 1991. The third edition was published this month in 2008. Yet the magnitude of errors in cost tracking systems is essentially the same today as it was in 1991. Following is an excerpt from the 3rd edition that summarizes the main issues of leakage from cost tracking systems:

"It is a regrettable fact that most corporate tracking systems for effort and costs (dollars, work hours, person months, etc.) are incorrect and manage to omit from 30% to more than 70% of the real effort applied to software projects. Thus most companies cannot safely use their own historical data for predictive purposes. When SPR personnel go on site and interview managers and technical personnel, these errors and omissions can be partially corrected by interviews.

The commonest omissions from historical data, ranked in order of significance, include the following:

Sources of Cost Errors	Magnitude of Cost Errors
1) Unpaid overtime by exempt staff	(up to 25% of reported effort)
2) Charging time to the wrong project	(up to 20% of reported effort)
3) User effort on software projects	(up to 20% of reported effort)
4) Management effort on software projects	(up to 15% of reported effort)
5) Specialist effort on software projects	(up to 15% of reported effort)
Human factors specialists	
Data base administration specialists	
Integration specialists	
Quality assurance specialists	
Technical writing specialists	
Education specialists	
Hardware or engineering specialists	
Marketing specialists	
Metrics and function point specialists	
6) Effort spent prior to cost tracking start up	(up to 10% of reported effort)
7) Inclusion/exclusion of non-project tasks	(up to 25% of reported effort)
Departmental meetings	
Courses and education	
Travel	
Overall Error Magnitude	(up to 125% of reported effort)

Not all of these errors are likely to occur on the same project, but enough of them occur so frequently that ordinary cost data from project tracking systems is essentially useless for serious economic study, for benchmark comparisons between companies, or for baseline analysis to judge rates of improvement.

A more fundamental problem is that most enterprises simply do not record data for anything but a small subset of the activities actually performed. In carrying out interviews with project managers and project teams to validate and correct historical data, the author and the consulting staff of SPR have observed the following patterns of incomplete and missing data, using the 25 activities of the standard SPR chart of accounts as the reference model:

Activities Performed	Completeness of historical data
01 Requirements	Missing or Incomplete
02 Prototyping	Missing or Incomplete
03 Architecture	Missing or Incomplete
04 Project planning	Missing or Incomplete
05 Initial analysis and design	Missing or Incomplete
06 Detail design	Incomplete
07 Design reviews	Missing or Incomplete
08 Coding	Complete
09 Reusable code acquisition	Missing or Incomplete
10 Purchased package acquisition	Missing or Incomplete
11 Code inspections	Missing or Incomplete
12 Independent verification and validation	Complete
13 Configuration management	Missing or Incomplete
14 Integration	Missing or Incomplete
15 User documentation	Missing or Incomplete
16 Unit testing	Incomplete
17 Function testing	Incomplete
18 Integration testing	Incomplete
19 System testing	Incomplete
20 Field testing	Missing or Incomplete
21 Acceptance testing	Missing or Incomplete
22 Independent testing	Complete
23 Quality assurance	Missing or Incomplete
24 Installation and training	Missing or Incomplete
25 Project management	Missing or Incomplete
26 Total project resources, costs	Incomplete

When the author and his colleagues collect data, we ask the managers and personnel to try and reconstruct any missing cost elements. Reconstruction of data from memory is plainly inaccurate, but it is better than omitting the missing data entirely. Unfortunately, the bulk of the software literature and many historical studies only report information to the level of complete projects, rather than to the level of specific activities. Such gross "bottom line" data cannot readily be validated and is almost useless for serious economic purposes."

To illustrate the effect of "leakage" from software tracking systems, consider what the complete development cycle would look like for a sample project. The sample is for a PBX switching system written in the C programming language. Table 1 illustrates a full set of activities and a full set of costs:

Table 1: Example of Complete Costs for Software Development

Average monthly salary =	$5,000
Burden rate =	50%
Fully burdened monthly rate =	$7,500
Work hours per calendar month =	132
Application size in FP =	1,500
Application type =	Systems
CMM level =	1
Programming lang. =	C
LOC per FP =	128

Activities	Staff Funct. Pt. Assignment Scope	Monthly Funct. Pt. Production Rate	Work Hours per Funct. Pt.	Burdened Cost per Funct. Pt.	Schedule Months	Staff	Effort Months
01 Requirements	500	200	0.66	$37.50	2.50	3.00	7.50
02 Prototyping	500	150	0.88	$50.00	3.33	3.00	10.00
03 Architecture	1,000	300	0.44	$25.00	3.33	1.50	5.00
04 Project Plans	1,000	500	0.26	$15.00	2.00	1.50	3.00
05 Initial Design	250	175	0.75	$42.86	2.86	3.00	8.57
06 Detail Design	250	150	0.88	$50.00	1.67	6.00	10.00
07 Design Reviews	200	225	0.59	$33.33	0.89	7.50	6.67
08 Coding	150	25	5.28	$300.00	6.00	10.00	60.00
09 Reuse acquisition	500	1,000	0.13	$7.50	0.50	3.00	1.50
10 Package purchase	2,000	2,000	0.07	$3.75	1.00	0.75	0.75
11 Code inspections	150	75	1.76	$100.00	2.00	10.00	20.00
12 Ind. Verif. & Valid.	1,000	250	0.53	$30.00	4.00	1.50	6.00
13 Configuration mgt.	1,500	1,750	0.08	$4.29	0.86	1.00	0.86
14 Integration	750	350	0.38	$21.43	2.14	2.00	4.29
15 User documentation	1,000	75	1.76	$100.00	13.33	1.50	20.00
16 Unit testing	200	150	0.88	$50.00	1.33	7.50	10.00
17 Function testing	250	150	0.88	$50.00	1.67	6.00	10.00
18 Integration testing	250	175	0.75	$42.86	1.43	6.00	8.57
19 System testing	250	200	0.66	$37.50	1.25	6.00	7.50
20 Field (Beta) testing	1,000	250	0.53	$30.00	4.00	1.50	6.00
21 Acceptance testing	1,000	350	0.38	$21.43	2.86	1.50	4.29
22 Independent testing	750	200	0.66	$37.50	3.75	2.00	7.50
23 Quality assurance	1,500	250	0.53	$30.00	6.00	1.00	6.00
24 Installation/training	1,500	250	0.53	$30.00	6.00	1.00	6.00

25 Project management	1,000	75	1.76	$100.00	13.33	1.50	20.00
Cumulative Results	420	6	22	$1,249.94	24.65	3.57	249.99

Now consider what the same project would look like if only coding and unit testing were recorded by the company's tracking system. Table 2 illustrates the results:

Table 2: Example of Partial Costs for Software Development

Average monthly salary =	$5,000
Burden rate =	50%
Fully burdened monthly rate =	$7,500
Work hours per calendar month =	132
Application size in FP =	1,500
Application type =	Systems
CMM level =	1
Programming lang. =	C
LOC per FP =	128

Activities	Staff Funct. Pt. Assignment Scope	Monthly Funct. Pt. Production Rate	Work Hours per Funct. Pt.	Burdened Cost per Funct. Pt.	Schedule Months	Staff	Effort Months
01 Coding	150	25	5.28	$300.00	6.00	10.00	60.00
02 Unit testing	200	150	0.88	$50.00	1.33	7.50	10.00
Cumulative Results	171	21	6.16	$350.00	6.60	8.75	70.00

Instead of a productivity rate of 6 function points per staff month, Table 2 indicates a productivity rate of 21 function points per staff month. Instead of a schedule of almost 25 calendar months, Table 2 indicates a schedule of less than 7 calendar months. Instead of a cost of $1,874, 910 the reported cost is only $525,000. Yet both Tables 1 and 2 are for exactly the same project. Unfortunately, what passes for "historical data" far more often matches the partial results shown in Table 2 than the complete results shown in Table 1.

Internal software projects where the development organization is defined as a cost center are the most incomplete and inaccurate in collecting software data. Many in-house projects by both corporations and government agencies lack useful historical data. Thus such organizations tend to very optimistic in their internal estimates because they have no solid basis for comparison. If they switch to a commercial estimating tool, they tend to be surprised at how much more costly the results might be.

External projects that are being built under contract, and projects where the development organization is a profit center, have stronger incentives to capture costs with accuracy. Thus contractors and outsource vendors are likely to keep better records than internal software groups.

Tables 1 and 2 show how wide the differences can be between full measurement and partial measurement. But an even wider range is possible, because many companies measure only coding, and don't record unit test as a separate cost element.

Table 3 shows the approximate distribution of tracking methods noted at more than 150 companies visited by the author:

Table 3: Distribution of Cost Tracking Methods

Activities	**Percent of Projects**
Coding only	25.00%
Coding, Unit test	25.00%
Design, Coding, and Unit test	15.00%
Requirements, Design, Coding, and Unit Test	10.00%
All development, but not Project Management	10.00%
All development and Project Management	15.00%
	100.00%

Leakage from cost tracking systems and the wide divergence in what activities are included presents a major problem to the software industry. It is very difficult to perform statistical analysis or create accurate benchmarks when so much of the reported data is incomplete, and there are so many variations in what gets recorded.

The gaps and variations in "historical data" explain why the author and his colleagues find it necessary to go on site and interview project managers and technical staff before accepting historical data. Unverified historical data is often so incomplete as to negate the value of using it for benchmarks and industry studies.

The problems illustrated by tables 1, 2, and 3 are just the surface manifestation of a deeper issue. After more than 50 years, the software industry lacks anything that

resembles a standard chart of accounts for collecting historical data. This lack is made more difficult by the fact that in real life, there are many variations of activities that are actually performed. There are variations due to application size, and variations due to application type. Consider both sets of patterns.

Variations in Software Development Activities by Size of the Application

In many industries building large products is not the same as building small products. Consider the differences in specialization and methods required to build a rowboat versus building an 80,000 ton cruise ship. A rowboat can be constructed by a single individual using only hand tools. But a large modern cruise ship requires more than 250 workers including many specialists such a pipe fitters, electricians, steel workers, painters, and even interior decorators. Software follows a similar pattern: Building large system in the 10,000 to 100,000 function point range is more or less equivalent to building other large structures such as ships, office buildings, or bridges. Many kinds of specialists are utilized and the development activities are quite extensive compared to smaller applications. Table 4 illustrates the variations in development activities noted for the six size plateaus using the author's 25-activity checklist for development projects:

Table 4: Development Activities for Six Project Size Plateaus

Activities Performed	1 Function Point	10 Function Points	100 Function Points	1000 Function Points	10,000 Function Points	100,000 Function Points
01 Requirements	X	X	X	X	X	X
02 Prototyping				X	X	X
03 Architecture					X	X
04 Project plans				X	X	X
05 Initial design		X	X	X	X	X
06 Detail design			X	X	X	X
07 Design reviews					X	X
08 Coding	X	X	X	X	X	X
09 Reuse acquisition	X	X	X	X	X	X
10 Package purchase					X	X
11 Code inspections				X	X	X
12 Ind. Verif. & Valid.						
13 Change control				X	X	X
14 Formal integration				X	X	X
15 User documentation			X	X	X	X
16 Unit testing	X	X	X	X	X	X
17 Function testing			X	X	X	X
18 Integration testing				X	X	X
19 System testing				X	X	X
20 Beta testing					X	X
21 Acceptance testing				X	X	X
22 Independent testing						
23 Quality assurance						X

24 Installation/training				X	X	X
25 Project management			X	X	X	X
Activities	4	5	9	18	22	23

Below the plateau of 1000 function points (which is roughly equivalent to 100,000 source code statements in a procedural language such as COBOL) less than half of the 25 activities are normally performed. But large systems in the 10,000 to 100,000 function point range perform more than 20 of these activities.

Variations in Software Development Activities by Type of Software

The second factor that exerts an influence on software development activities is the type of software being constructed. For example, the methods utilized for building military software is very different from civilian norms. The systems and commercial software domains also have fairly complex development activities compared to management information systems. The outsource domain, due to contractual implications, also uses a fairly extensive set of development activities.

Table 5 illustrates the differences in development activities that the author has noted across the six types of software:

Table 5: Development Activities for Six Project Types

Activities Performed	Web	MIS	Outsource	Commercial	Systems	Military
01 Requirements		X	X	X	X	X
02 Prototyping	X	X	X	X	X	X
03 Architecture			X	X	X	X
04 Project plans		X	X	X	X	X
05 Initial design		X	X	X	X	X
06 Detail design		X	X	X	X	X
07 Design reviews			X	X	X	X
08 Coding	X	X	X	X	X	X
09 Reuse acquisition	X	X	X	X	X	X
10 Package purchase	X	X	X	X	X	X
11 Code inspections			X	X	X	X
12 Ind. Verif. & Valid.						X
13 Change control		X	X	X	X	X
14 Formal integration		X	X	X	X	X
15 User documentation		X	X	X	X	X
16 Unit testing	X	X	X	X	X	X
17 Function testing		X	X	X	X	X
18 Integration testing		X	X	X	X	X
19 System testing	X	X	X	X	X	X
20 Beta testing				X	X	X
21 Acceptance testing		X	X	X	X	X
22 Independent testing						X

23 Quality assurance			X	X	X	X
24 Installation/training		X	X	X	X	X
25 Project management		X	X	X	X	X
Activities	6	18	22	23	23	25

As can be seen, the activities for outsourced, commercial, systems, and military software are somewhat more numerous than for MIS projects where development processes tend to be rudimentary in many cases.

Development Activities for Projects Using the CMM and Agile Methods

Two of the most popular development approaches are those based on the famous capability maturity model (CMM) developed by the Software Engineering Institute (SEI) in 1987 and the newer "Agile" approach first published in 2001.

There is now fairly solid evidence about the CMM from many studies. When organizations move from CMM level 1 up to level 2, 3, 4, and 5 their productivity and quality levels tend to improve based on samples at each level. When they adopt the newer Team Software Process (TSP) and Personal Software Process (PSP) there is an additional boost in performance.

As of 2007 the newer CMMI has less empirical data than the older CMM, which is not surprising given the more recent publication of the CMMI. However the TSP and PSP methods do have enough data to show that they are successful in improving both quality and productivity at the same time.

Since the CMM originated in the 1980's the "waterfall" method of development was the most common at that time and was implicitly supported by the early CMM. However other methods such as spiral, iterative, etc. were quickly included in the CMM as well.

What the CMM provided was a solid framework of activities, much better rigor in the areas of quality control and change management, and much better measurement of progress, quality, and productivity than was previously the norm.

Measurement and collection of data for projects that use the CMM or CMMI tend to be fairly complete. In part this is due to the measurement criteria of the CMM and CMMI, and in part it is due to the fact that many projects using the CMM and CMMI are contract projects where accurate time and expense records are required under the terms of the contracts.

Watt Humphrey's newer TSP and PSP are also very good in collecting data. Indeed the TSP and PSP data is among the most precise ever collected. However the TSP data is collected using "task hours" or the actual number of hours for specific tasks. Non-task activities such as departmental meetings and training classes are excluded.

The history of the Agile methods is not as clear as the history of the CMM, because the Agile methods are somewhat diverse. However in 2001 the famous "Agile manifesto" was published. This provided the essential principles of Agile development. That being said, there are quite a few Agile variations include Extreme Programming (XP), Crystal Development, Adaptive Software Development, Feature Driven Development, and several others.

Some of the principal beliefs found in the Agile manifesto include:

- Working software is the goal, not documents
- Working software is the primary measure of success
- Close and daily contact between developers and clients are necessary
- Face to face conversation is the best form communication
- Small self-organizing teams give the best results
- Quality is critical, so testing should be early and continuous

The Agile methods, the CMM, and the CMMI are all equally concerned about three of the same fundamental problems:

1. Software requirements always change.
2. Fixing software bugs is the most expensive software activity in history.
3. High quality leads to high productivity and short schedules.

However the Agile method and the CMM/CMMI approach draw apart on two other fundamental problems:

4. Paperwork is the second most expensive software activity in history.
5. Without careful measurements continuous progress is unlikely.

The Agile methods take a strong stand that paper documents in the form of rigorous requirements and specifications are too slow and cumbersome to be effective. In the Agile view, daily meetings with clients are more effective than written specifications. In the Agile view, daily team meetings or "Scrum" sessions are the best way of tracking progress, as opposed to written status reports. The CMM and CMMI do not fully endorse this view.

The CMM and CMMI take a strong stand that measurements of quality, productivity, schedules, costs, etc. are a necessary adjunct to process improvement and should be done well. In the view of the CMM and CMMI, without data that demonstrates effective progress it is hard to prove that a methodology is a success or not. The Agile methods do not fully endorse this view. In fact, one of the notable gaps in the Agile approach is any quantitative quality or productivity data that can prove the success of the methods.

While some Agile projects to measure, they often use metrics other than function points. For example some Agile projects use "story points" and others may use "web-object points" or "running tested features" (RTF). These metrics are interesting, but lack large

collections of historical data and therefore cannot easily be used for comparisons to older projects.

Differences in Development Activities between the Agile and CMM/CMMI Methods

Because the CMM approach was developed in the 1980's when the waterfall method was common, it is not difficult to identify the major activities that are typically performed. For an application of 1500 function points (approximately 150,000 source code statements) the following 20 activities would be typical using either the CMM or CMMI:

Table 6: Normal CMM and CMMI Activities for a Civilian Application of 1500 Function Points

1. Requirements
2. Prototyping
3. Architecture
4. Project planning and estimating
5. Initial design
6. Detailed design
7. Design inspections
8. Coding
9. Reuse acquisition
10. Code inspections
11. Change and configuration control
12. Software quality assurance
13. Integration
14. Test plans
15. Unit testing
16. New function testing
17. Regression testing
18. Integration testing
19. Acceptance testing
20. Project management

Using the CMM and CMMI the entire application of 1500 function points would have the initial requirements gathered and analyzed, the specifications written, and various planning document produced before coding got underway.

By contrast, the Agile methods of development would follow a different pattern. Because of the Agile goal is to deliver running and usable software to clients as rapidly as possible, the Agile approach would not wait for the entire 1500 function points to be designed before coding started.

What would be most likely with the Agile methods would be to divide the overall project into four smaller projects, each of about 300 function points in size. (Possibly as many as five subset projects might be used for 1500 function points.) In the Agile terminology these smaller segments are termed iterations or sometimes "sprints."

These subset iterations or sprints are normally developed in a "time box" fashion that ranges between perhaps 2 weeks and 3 months based on the size of the iteration. For the example here, we can assume about two calendar months for each iteration or sprint.

However in order to know what the overall general set of features would be, an Agile project would start with "Iteration 0" or a general planning and requirements-gathering session. At this session the users and developers would scope out the likely architecture of the application and then subdivide it into a number of iterations.

Also, at the end of the project when all of the iterations have been completed, it will be necessary to test the combined iterations at the same time. Therefore a release phase follows the completion of the various iterations. For the release, some additional documentation may be needed. Also, cost data and quality data needs to be consolidated for all of the iterations. A typical Agile development pattern might resemble the following:

Table 7: Normal Agile Activities for an Application of 1000 Function Points

Iteration 0
1. General overall requirements
2. Planning
3. Sizing and estimating
4. Funding

Iterations 1-4
1. User requirements for each iteration
2. Test planning for each iteration
3. Testing case development for each iteration
4. Coding
5. Testing
6. Scrum sessions
7. Iteration documentation
8. Iteration cost accumulation
9. Iteration quality data

Release
1. Integration of all iterations
2. Final testing of all iterations
3. Acceptance testing of application
4. Total cost accumulation
5. Quality data accumulation
6. Final scrum session

The most interesting and unique features of the Agile methods are these: 1) The decomposition of the application into separate iterations; 2) The daily face-to-face contact with one or more user representatives; 3) The daily "scrum" sessions to discuss

the backlog of work left to be accomplished and any problems that might slow down progress. Another interesting feature is to create the test cases before the code itself is written, which is a feature of Extreme programming (XP) and several other Agile variations.

The Hazards and Errors of "Phase Level" Measurements

Another weakness of software measurement is the chart of accounts used, or the set of activities for which resource and cost data are collected. The topic of selecting the activities to be included in software project measurements is a difficult issue and cannot be taken lightly. There are five main contenders:

1) Project-level measurements
2) Phase-level measurements
3) Activity-level measurements
4) Task-level measurements
5) Subtask-level measurements

Project-level measurements and phase-level measurements have been the most widely used for more than 50 years, but they are also the least accurate. Of these five, only activity, task, and subtask measurements will allow data collection with a precision of better than 10% and support the concept of activity-based costing. Neither project level nor phase level data will be useful in exploring process improvements, or in carrying out multiple regression analysis to discover the impact of various tools, methods, and approaches. Collecting data only at the level of projects and phases correlates strongly with failed or canceled measurement programs, since the data cannot be used for serious process research.

Historically, project-level measurements have been used most often. Phase-level measurements have ranked second to project-level measurements in frequency of usage. Unfortunately phase-level measurements are inadequate for serious economic study. Many critical activities such as "user documentation" or "formal inspections" span multiple phases and hence tend to be invisible when data is collected at the phase level.

Also, data collected at the levels of activities, tasks, and subtasks can easily be rolled up to provide phase-level and project-level views. The reverse is not true: you cannot explode project-level data or phase-level data down to the lower levels with acceptable accuracy and precision. If you start with measurement data that is too coarse, you will not be able to do very much with it.

Table 8 gives an illustration that can clarify the differences. Assume you are thinking of measuring a project such as the construction of a small PBX switching system used earlier in this paper. Here are the activities that might be included at the level of the project, phases, and activities for the chart of accounts used to collect measurement data:

Table 8: Project, Phase, and Activity-Level Measurement Charts of Accounts

Project Level	Phase Level	Activity Level
PBX switch	1) Requirements	1) Requirements
	2) Analysis	2) Prototyping
	3) Design	3) Architecture
	4) Coding	4) Planning
	5) Testing	5) Initial design
	6) Installation	6) Detail design
		7) Design review
		8) Coding
		9) Reused code acquisition
		10) Package acquisition
		11) Code inspection
		12) Independent verif. & validation
		13) Configuration control
		14) Integration
		15) User documentation
		16) Unit test
		17) Function test
		18) Integration test
		19) System test
		20) Field test
		21) Acceptance test
		22) Independent test
		23) Quality assurance
		24) Installation
		25) Management

If you collect measurement cost data only to the level of a project, you will have no idea of the inner structure of the work that went on. Therefore the data will not give you the ability to analyze activity-based cost factors, and is almost useless for purposes of process improvement. This is one of the commonest reasons for the failure of software measurement programs: the data is not granular enough to find out why projects were successful or unsuccessful.

Measuring at the phase level is only slightly better. There are no standard phase definitions nor any standards for the activities included in each phase. Worse, activities such as "project management" which span every phase are not broken out for separate cost analysis. Many activities span multiple phases, so phase-level measurements are not effective for process improvement work.

Measuring at the activity level does not imply that every project performs every activity. For example, small MIS projects and client-server applications normally perform only 10

or so of the 25 activities that are shown above. Systems software such as operating systems and large switching systems will typically perform about 20 of the 25 activities. Only large military and defense systems will routinely perform all 25. Here too, by measuring at the activity level useful information becomes available. It is obvious that one of the reasons that systems and military software have much lower productivity rates than MIS projects is because they do many more activities for a project of any nominal size.

Measuring at the task and sub-task levels are more precise than activity-level measurements but also much harder to accomplish. However in recent years Watts Humphrey's Team Software Process (TSP) and Personal Software Process (PSP) have started accumulating effort data to the level of tasks. This is perhaps the first time that such detailed information has been collected on a significant sample of software projects.

Variations in Burden Rates or Overhead Costs

A major problem associated with software cost studies is the lack of generally accepted accounting practices for determining the burden rate or overhead costs that are added to basic salaries to create a metric called "the fully burdened salary rate" which corporations use for determining business topics such as the charge-out rates for cost centers. The fully burdened rate is also used for other business purposes such as contracts, outsource agreements, and return on investment (ROI) calculations.

The components of the burden rate are highly variable from company to company. Some of the costs included in burden rates can be: social security contributions, unemployment benefit contributions, various kinds of taxes, rent on office space, utilities, security, postage, depreciation, portions of mortgage payments on buildings, various fringe benefits (medical plans, dental plans, disability, moving and living, vacations, etc.), and sometimes the costs of indirect staff (human resources, purchasing, mail room, etc.)

One of the major gaps in the software literature, and for that matter in accounting literature, is the almost total lack of international comparisons of the typical burden rate methodologies used in various countries. So far as can be determined, there are no published studies that explore burden rate differences between countries such as the United States, Canada, India, the European Union countries, Japan, China, etc.

Among the author's clients, the range of burden rates runs from a low of perhaps 15% of basic salary levels to a high of approximately 300%. In terms of dollars, that range means that the fully-burdened charge rate for the position of senior systems programmer in the United States can run from a low of about $15,000 per year to a high of $350,000 per year.

Unfortunately, the software literature is almost silent on the topic of burden or overhead rates. Indeed, many of the articles on software costs not only fail to detail the factors included in burden rates, buy often fail to even state whether the burden rate itself was used in deriving the costs that the articles are discussing!

Table 9 illustrates some of the typical components of software burden rates, and also how these components might vary between a large corporation with a massive infrastructure and a small start-up corporation that has very few overhead cost elements.

Table 9: Components of Typical Burden Rates in Large and Small Companies

Large Company			**Small Company**		
Average annual salary	$50,000	100.0%	Average annual salary	$50,000	100.0%
Personnel Burden			**Personnel Burden**		
Payroll taxes	$5,000	10.0%	Payroll taxes	$5,000	10.0%
Bonus	$5,000	10.0%	Bonus	0	0.0%
Benefits	$5,000	10.0%	Benefits	$2,500	5.0%
Profit sharing	$5,000	10.0%	Profit sharing	0	0.0%
Subtotal	*$20,000*	*40.0%*	*Subtotal*	*$7,500*	*15.0%*
Office Burden			**Office Burden**		
Office rent	$10,000	20.0%	Office rent	$5,000	10.0%
Property taxes	$2,500	5.0%	Property taxes	$1,000	2.0%
Office supplies	$2,000	4.0%	Office supplies	$1,000	2.0%
Janitorial service	$1,000	2.0%	Janitorial service	$1,000	2.0%
Utilities	$1,000	2.0%	Utilities	$1,000	2.0%
Subtotal	*$16,500*	*33.0%*	*Subtotal*	*$9,000*	*18.0%*
Corporate Burden			**Corporate Burden**		
Information Systems	$5,000	10.0%	Information Systems	0	0.0%
Finance	$5,000	10.0%	Finance	0	0.0%
Human Resources	$4,000	8.0%	Human Resources	0	0.0%
Legal	$3,000	6.0%	Legal	0	0.0%
Subtotal	*$17,000*	*34.0%*	*Subtotal*	*0*	*0.0%*
Total Burden	$53,500	107.0%	Total Burden	$16,500	33.0%
Salary + Burden	$103,500	207.0%	Salary + Burden	$66,500	133.0%
Monthly rate	$8,625		Monthly rate	$5,542	

When the combined ranges of basic salaries and burden rates are applied to software projects in the United States, they yield almost a 6 to 1 variance in billable costs for projects where the actual number of work months or work hours are identical!

When the salary and burden rate ranges are applied to international projects, they yield about a 15 to 1 variance between countries such as India or China at the low end of the spectrum, and Germany or Switzerland or Japan on the high end of the spectrum.

Hold in mind that this 15 to 1 range of cost variance is for projects where the actual number of hours worked is identical. When productivity differences are considered too, there is more than a 100 to 1 variance between the most productive projects in companies with the lowest salaries and burden rates and the least productive projects in companies with the highest salaries and burden rates.

Variations in Costs by Industry

Although software productivity measurements based on human effort in terms of work hours or work months can be measured with acceptable precision, the same cannot be said for software costs.

A fundamental problem with software cost measures is the fact that salaries and compensation vary widely from job to job, worker to worker, company to company, region to region, industry to industry, and country to country.

Among the author's clients in the United States the basic salary of the occupation of "software project manager" ranges from a low of about $42,000 per year to a high of almost $125,000 per year. When international clients are included, the range for the same position runs from less than $10,000 per year to more than $170,000 a year.

Table 10 shows averages and ranges for project management compensation for 20 U.S. industries. Table 10 shows average results in the left column, and then shows the ranges observed based on company size, and on geographic region. In general, large corporations pay more than small companies. For example, large urban areas such as the San Francisco Bay area or the urban areas in New York and New Jersey have much higher pay scales than do more rural areas or smaller communities.

Also some industries such as banking and financial services and telecommunications manufacturing tend to have compensation levels that are far above U.S. averages, while other industries such as government service and education tend to have compensation levels that are significantly lower than U.S. averages.

These basic economic facts mean that it is unsafe and inaccurate to use "U.S. averages" for cost comparisons of software. At the very least, cost comparisons should be within the context of the same or related industries, and comparisons should be made against organizations of similar size and located in similar geographic areas.

Table 10: Annual Salary Levels for Software Project Managers in 20 Industries in the United States

Industry	Average Annual Salary	Range by Company Size (+ or -)	Range by Geographic Region (+ or -)	Maximum Annual Salary	Minimum Annual Salary
Banking	$87,706	$20,000	$6,000	$113,706	$61,706
Electronics	$86,750	$15,000	$6,000	$107,750	$65,750
Telecommunications	$86,500	$15,000	$6,000	$107,500	$65,500
Software	$86,250	$15,000	$6,000	$107,250	$65,250
Consumer products	$85,929	$14,000	$5,500	$105,429	$66,429
Chemicals	$84,929	$13,000	$5,500	$103,429	$56,429
Defense	$80,828	$13,000	$5,500	$99,328	$62,328
Food/beverages	$78,667	$12,000	$5,000	$95,667	$61,667
Media	$75,125	$12,000	$5,000	$92,125	$58,125
Industrial equipment	$75,009	$12,000	$5,000	$92,009	$58,009
Distribution	$74,900	$11,000	$5,000	$90,900	$58,900
Insurance	$73,117	$10,000	$5,000	$88,117	$58,117
Public utilities	$71,120	$7,500	$4,500	$83,120	$59,120
Retail	$70,125	$7,000	$4,500	$81,625	$58,625
Health care	$67,880	$7,500	$4,500	$79,880	$55,880
Nonprofits	$66,900	$7,500	$4,500	$78,900	$54,900
Transportation	$66,500	$7,000	$4,500	$78,000	$55,000
Textiles	$65,585	$7,000	$4,500	$77,085	$54,085
Government	$63,150	$6,000	$4,000	$73,150	$53,150
Education	$62,375	$6,000	$4,000	$72,375	$52,375
Average	$75,467	$10,875	$5,025	$91,367	$59,567

Industry differences and differences in geographic regions and company sizes are so important that cost data cannot be accepted at face value without knowing the details of the industry, city, and company size.

Variations in Costs by Occupation Group

Other software-related positions besides project management have broad ranges in compensation too, and there are now more than 75 software-related occupations in the United States. This means that in order to do software cost studies it is necessary to deal with major differences in costs based on industry, on company size, and on geographic location, on the kinds of specialists that are present on any given project, and on years of tenure or merit appraisal results.

Table 11 illustrates the ranges of basic compensation (exclusive of bonuses or merit appraisal adjustments) for 15 software occupations in the United States. As can be seen,

the range of possible compensation levels runs from less than $30,000 to more than $100,000.

Over and above the basic compensation levels shown in table 11, a number of specialized occupations are now offering even higher compensation levels than those illustrated. For example, programmers who are familiar with SAP R/3 integrated system and the ABAP programming language can expect compensation levels about 10% higher than average, and may even receive a "signing bonus" similar to those offered to professional athletes.

Table 11: Variations in Compensation For 15 U.S. Software Occupation Groups

Occupation	Average Annual Salary	Range by Company Size (+ or -)	Range by Geographic Region (+ or -)	Range by Industry (+ or -)	Maximum Annual Salary	Minimum Annual Salary
Software Architect	$91,000	$13,000	$4,500	$7,500	$116,000	$66,000
Senior Systems Prog.	$90,000	$13,000	$4,500	$6,000	$113,500	$66,500
Senior Systems Anal.	$83,000	$11,000	$4,000	$6,000	$104,000	$62,000
Systems Programmer	$78,000	$12,000	$4,000	$5,500	$89,500	$56,500
Systems Analyst	$70,000	$10,500	$3,750	$5,000	$89,250	$50,750
Process Analyst	$67,000	$10,500	$3,750	$5,000	$86,250	$47,750
Programmer/analyst	$66,000	$11,000	$3,500	$5,000	$85,500	$46,500
Database analyst	$65,000	$12,000	$3,750	$6,000	$86,750	$43,250
Applic. Programmer	$65,000	$10,000	$3,500	$5,000	$83,500	$46,500
Mainten. Programmer	$63,000	$10,000	$3,500	$5,000	$81,500	$44,500
Testing Specialist	$62,500	$10,000	$3,500	$5,000	$81,000	$44,000
Metrics specialist	$62,000	$8,000	$3,750	$5,000	$78,750	$45,250
Quality Assurance	$61,000	$7,500	$3,500	$5,000	$77,000	$45,000
Technical writer	$51,500	$5,000	$3,500	$3,000	$53,000	$40,000
Customer support	$48,500	$2,000	$3,500	$2,000	$56,000	$41,000
Average	$68,233	$9,700	$3,767	$5,067	$86,767	$49,700

Even if only basic compensation is considered, it can easily be seen that software projects developed by large companies in large cities such as New York and San Francisco will have higher cost structures than the same applications developed by small companies in smaller cities such as Little Rock or Knoxville.

Although the topic is not illustrated and the results are often proprietary, there are also major variations in compensation based on merit appraisals and or longevity within grade. This factor can add about another plus or minus $7500 to the ranges of compensation for technical positions, and even more for executive and managerial positions.

Also not illustrated are the bonus programs and stock equity programs which many companies offer to software technical employees and to managers. For example the stock

equity program at Microsoft has become famous for creating more millionaires than any similar program in U.S. industry.

Variations in Work Habits and Unpaid Overtime

The software industry is highly labor-intensive one. So long as software is built using human effort as the primary tool, all of the factors associated with work patterns and overtime will continue to be significant. That being said, unpaid overtime is the most common omission from software cost tracking systems. Unpaid overtime averages more than 10 hours a week in the United States and more than 16 hours a week in Japan. This is far too significant a factor to be ignored, but that is usually the case.

Assume that a typical month contains four work weeks, each comprised of five eight-hour working days. The combination of 4 weeks * 5 days * 8 hours = 160 available hours in a typical month. However, at least in the United States, the *effective* number of hours worked each month is often less than 160, due to factors such as coffee breaks, meetings, slack time between assignments, and the like.

Thus in situations where there is no intense schedule pressure, the effective number of work hours per month may only amount to about 80% of the available hours, or about 128 hours per calendar month.

On the other hand, software projects are often under intense schedule pressures and overtime is quite common. The majority of professional U.S. software personnel are termed "exempt" which means that they do not receive overtime pay for work in the evening or on weekends. Indeed, many software cost tracking systems do not even record overtime hours.

Thus for situations where schedule pressures are intense, not only might the software team work the available 160 hours per month, but they would also work late in the evenings and on weekends too. Thus on "crunch" projects the work might amount to 110% of the available hours or about 176 hours per week.

Table 12 compares two versions of the same project, which can be assumed to be a 1000 function points information systems application written in COBOL. The first version is a "normal" version where only about 80% of the available hours each month are worked. The second version shows the same project in "crunch" mode where the work hours comprise 110%, with all of the extra hours being in the form of unpaid overtime by the software team.

Since exempt software personnel are normally paid on a monthly basis rather than an hourly basis, the differences in apparent results between "normal" and "intense" work patterns are both significant and also tricky when performing software economic analyses.

Table 12: Differences Between "Normal" and "Intense" Software Work Patterns

Activity	Project 1	Project 2	Difference	Percent
Work Habits	**Normal**	**Intense**		
Function Point Size	1000	1000	0	0.00%
Size in Lines of Code	100000	100000	0	0.00%
LOC per FP	100	100	0	0.00%
Ascope in FP	200	200	0	0.00%
Nominal Prate in FP	10	10	0	0.00%
Availability	*80.00%*	*110.00%*	*30.00%*	*37.50%* **
Hours per Month	*128.00*	*176.00*	*48.00*	*37.50%* **
Salary per Month	$5,000.00	$5,000.00	$0.00	0.00%
Staff	5.00	5.00	0.00	0.00%
Effort Months	125.00	90.91	-34.09	-27.27%
Schedule Months	31.25	16.53	-14.72	-47.11%
Cost	$625,000	$454,545	-$170,455	-27.27%
Cost per FP	$625.00	$454.55	-$170.45	-27.27%
Work Hours per FP	16.00	16.00	0.00	0.00%
Virtual Prate in FP	8.00	11.00	3.00	37.50%
Cost per LOC	$6.25	$4.55	-$1.70	-27.27%
LOC per Month	800	1100	300	37.50%

As can be seen from table 12, applying intense work pressure to a software project in the form of unpaid overtime can produce significant and visible reductions in software costs and software schedules. (However there may also be invisible and harmful results in terms of staff fatigue and burnout.)

Table 12 introduces five terms that are significant in software measurement and also cost estimating, but which need definition.

The first term is "Assignment scope" (abbreviated to Ascope) which is the quantity of function points normally assigned to one staff member.

The second term is "Production rate" (abbreviated to Prate) which is the monthly rate in function points at which the work will be performed.

The third term is "Nominal production rate" (abbreviated to Nominal Prate in FP) which is the rate of monthly progress measured in function points without any unpaid overtime being applied.

The fourth term is "Virtual production rate" (abbreviated to Virtual Prate in FP) which is the apparent rate of monthly productivity in function points that will result when unpaid overtime is applied to the project or activity.

The fifth term is "Work hours per function point" which simply accumulates the total number of work hours expended and divides that amount by the function point total of the application.

Because software staff members are paid monthly but work hourly, the most visible impact of unpaid overtime is to decouple productivity measured in work hours per function point from productivity measured in function points per staff month.

Assume that a small 60 function point project would normally require two calendar months or 320 work hours to complete. Now assume that the programmer assigned worked double shifts and finished the project in one calendar month, although 320 hours were still needed.

If the project had been a normal one stretched over two months, the productivity rate would have been 30 function points per staff month and 5.33 work hours per function point. By applying unpaid overtime to the work and finishing in one month, the virtual productivity rate appears to be 60 function points per staff month, but the actual number of hours required remains 5.33 work hours per function points.

Variations in work patterns are extremely significant variation when dealing with international software projects. There are major national differences in terms of work hours per week, quantities of unpaid overtime, numbers annual holidays, and annual vacation periods.

In fact, it is very dangerous to perform international studies without taking this phenomenon into account. Variations in work practices are a major differentiating factor for international software productivity and schedule results.

Table 13 makes a number of simplifying assumptions and does not deal with the specifics of sick time, lunch and coffee breaks, meetings, courses, and non-work activities that might occur during business hours.

Table 13 is derived from basic assumptions about national holidays and average annual vacation times. Table 13 also ignores telecommuting, home offices, flex time, and a number of other factors that are important for detailed analyses.

Since there are significant local and industry differences within every country, the data in Table 13 should be used just as a starting point for more detailed exploration and analysis:

Table 13: Approximate Number of Work Hours per Year in 10 Countries

Country	Work days per year	Work hours per day	Overtime per day	Work hours per year	Percent of U.S. Results
Japan	260	9.00	2.5	2,990	139%
China	260	9.00	1.5	2,730	127%
India	245	8.50	2	2,573	120%
Italy	230	9.00	1	2,300	107%
United States	239	8.00	1	2,151	100%
Brazil	234	8.00	1	2,106	98%
United Kingdom	232	8.00	1	2,088	97%
France	230	8.00	1	2,070	96%
Germany	228	8.00	0	1,824	85%
Russia	230	7.50	0	1,725	80%
AVERAGE	238.8	8.30	1.1	2,245	104%

Software is currently among the most labor-intensive commodities on the global market. Therefore work practices and work effort applied to software exerts a major influence on productivity and schedule results. In every country, the top software personnel tend to work rather long hours so table 9 can only be used for very rough comparisons.

The differences in national work patterns compounded with differences in burdened cost structures can lead to very significant international differences in software costs and schedules for the same size and kind of application.

Variations in Costs by Nature, Scope, Class, Type, and Complexity of Applications

In comparing one software project against another, it is important to know exactly what kinds of software applications are being compared. This is not as easy as it sounds. The industry lacks a standard taxonomy of software projects that can be used to identify projects in a clear and unambiguous fashion.

Since 1985 the author has been using a multi-part taxonomy for classifying projects. The major segments of this taxonomy include nature, scope, class, type, and complexity. Following are the basic definitions of the author's taxonomy:

PROJECT NATURE: _____

1. New program development
2. Enhancement (new functions added to existing software)
3. Maintenance (defect repair to existing software)
4. Conversion or adaptation (migration to new platform)
5. Reengineering (re-implementing a legacy application)
6. Package modification (revising purchased software)

PROJECT SCOPE: _____

1. Subroutine
2. Module
3. Reusable module
4. Disposable prototype
5. Evolutionary prototype
6. Standalone program
7. Component of a system
8. Release of a system (other than the initial release)
9. New system (initial release)
10. Enterprise system (linked integrated systems)

PROJECT CLASS: _____

1. Personal program, for private use
2. Personal program, to be used by others
3. Academic program, developed in an academic environment
4. Internal program, for use at a single location
5. Internal program, for use at a multiple locations
6. Internal program, for use on an intranet
7. Internal program, developed by external contractor
8. Internal program, with functions used via time sharing
9. Internal program, using military specifications
10. External program, to be put in public domain
11. External program to be placed on the Internet
12. External program, leased to users
13. External program, bundled with hardware
14. External program, unbundled and marketed commercially
15. External program, developed under commercial contract
16. External program, developed under government contract
17. External program, developed under military contract

PROJECT TYPE: _____

1. Nonprocedural (generated, query, spreadsheet)
2. World Wide Web application
3. Batch applications program
4. Interactive applications program
5. Interactive GUI applications program
6. Batch database applications program
7. Interactive database applications program
8. Client/server applications program
9. Scientific or mathematical program
10. Systems or support program including "middleware"
11. Communications or telecommunications program
12. Process-control program
13. Trusted system
14. Embedded or real-time program
15. Graphics, animation, or image-processing program
16. Multimedia program
17. Robotics, or mechanical automation program
18. Artificial intelligence program
19. Neural net program
20. Hybrid project (multiple types)

PROBLEM COMPLEXITY: _____

1. No calculations or only simple algorithms
2. Majority of simple algorithms and simple calculations
3. Majority of simple algorithms plus a few of average complexity
4. Algorithms and calculations of both simple and average complexity
5. Algorithms and calculations of average complexity
6. A few difficult algorithms mixed with average and simple
7. More difficult algorithms than average or simple
8. A large majority of difficult and complex algorithms
9. Difficult algorithms and some that are extremely complex
10. All algorithms and calculations are extremely complex

CODE COMPLEXITY: _____

1. Most "programming" done with buttons or pull down controls
2. Simple nonprocedural code (generated, database, spreadsheet)
3. Simple plus average nonprocedural code
4. Built with program skeletons and reusable modules
5. Average structure with small modules and simple paths
6. Well structured, but some complex paths or modules
7. Some complex modules, paths, and links between segments
8. Above average complexity, paths, and links between segments
9. Majority of paths and modules are large and complex
10. Extremely complex structure with difficult links and large modules

DATA COMPLEXITY: _____

1. No permanent data or files required by application
2. Only one simple file required, with few data interactions
3. One or two files, simple data, and little complexity
4. Several data elements, but simple data relationships
5. Multiple files and data interactions of normal complexity
6. Multiple files with some complex data elements and interactions
7. Multiple files, complex data elements and data interactions
8. Multiple files, majority of complex data elements and interactions
9. Multiple files, complex data elements, many data interactions
10. Numerous complex files, data elements, and complex interactions

In addition, the author also uses codes for country (telephone codes work for this purpose as do ISO country codes), for industry (Department of Commerce North American Industry Classifications (NAIC)), and geographic region (Census Bureau state codes work in the United States. Five-digit zip codes or telephone area codes could also be used.)

Why such a complex multi-layer taxonomy is necessary can be demonstrated by a thought experiment of comparing the productivity rates of two unlike applications. Suppose the two applications have the following aspects:

Parameter	Application A	Application B
Country code	1 = United States	1 = United States
Region code	06 = California	06 = California
Industry code	1569 = telecommunications	2235 = National Security
Nature	1 = New	1 = New
Scope	4 = Disposable prototype	9 = New system
Class	1 = Personal program	17 = Military contract
Type	1 = Non procedural	13 = Trusted system
Problem complexity	1 = Simple	9 = Complex algorithms
Code complexity	2 = Non procedural code	10 = Highly complex code
Data complexity	2 = No files needed	10 = Many complex files

The productivity rate of Application A can very well exceed the productivity rate of Application B by more than two orders of magnitude. The absolute amount of effort devoted to Project B can exceed the effort devoted to Project A by more than 1000 to 1. The cost per function point for Application A will probably be less than $250 per function point, while Application B will probably cost more than $3000 per function point.

These two examples are from the same country, geographic region, but different industry segments.. If one of the projects were done in China or India, the ranges would be even broader by another 200% or so. If a high-cost country such as Switzerland was one of the locations, the costs would swing upwards.

Does that mean that the technologies, tools, or skills on Project A are superior to those used on Project B? It does not -- it simply means two very different kinds of software project are being compared, and great caution must be used to keep from drawing incorrect conclusions.

In particular, software tool and methodology vendors should exercise more care when developing their marketing claims, many of which appear to be derived exclusively from comparisons of unlike projects in terms of the nature, scope, class, type, and size parameters. Several times vendors have made claims of “10 to 1” improvements in productivity rates as a result of using their tools. Invariably these claims are false, and are based on comparison of unlike projects. The most common error is that of comparing a very small project against a very large system. Another common error is to compare only a subset of activities, such as coding, against a complete project measured from requirement to delivery.

Only last week the author received a question from a webinar attendee about object-oriented productivity rates. The questioner said that his group using OO programming languages was more than 10 times as productive as the data cited in the webinar. But the data the questioner was using consisted only of code development. The data in the webinar ran from requirements to delivery and also included project management.

Unfortunately not using a standard taxonomy and not clearly identifying the activities that are included in the data is the norm for software measurements circa 2008.

The Missing Links of Measurement: When Do Projects Start? When do they End?

When a major software project *starts* is the single most ambiguous point in the entire life cycle. For many projects, there can be weeks or even months of informal discussions and preliminary requirements gathering before it is decided that the application looks feasible. (If the application does not look feasible and no project results, there may still be substantial resources expended that it would be interesting to know about.) Even when it is decided to go forward with the project, that does not automatically imply that the decision was reached on a particular date which can be used to mark the commencement of billable or formal work.

So far as can be determined there are no standards or even any reasonable guidelines for determining the exact starting points of software projects. The methodology used by the author to determine project starting dates is admittedly crude: I ask the senior project manager for his or her opinion as to when the project began, and utilize that point unless there is a better source.

Sometimes a formal request for proposal (RFP) exists, and also the responses to the request. For contract projects, the date of the signing of the contract may serve as the starting point. However, for the great majority of systems and MIS applications, the exact starting point is clouded in uncertainty and ambiguity.

Although the end date of a software project is less ambiguous than the start date, there are still many variances. The author uses the delivery of the software to the first actual customer as the termination date or the end of the project. While this works for commercial packages such as Microsoft Office, it does not work for major applications such as ERP packages where the software cannot be used on the day of delivery, but instead has to be installed and customized for each specific client. In this situation, the actual end of the project would be the date of the initial usage of the application for business purposes. In the case of major ERP packages such as SAP and Oracle, successfully using the software can be more than 12 months after the software was delivered and installed on the customer's computers.

The bottom line is that the software industry as of 2008 does not have any standard method or unambiguous methods for measuring either the start or end dates of major software projects. As a result, "historical data" on schedules is highly suspect.

Gaps and Errors in Measuring Schedule Overlap and Schedule Slippage

Since software project schedules are among the most critical of all software project factors, one might think that methods for measuring schedules would be fully matured after some 50 years of trial and error. There is certainly no shortage of project scheduling tools, many of which can record historical data as well as plan unfinished projects.

However, the measurement of original schedules, slippages to those schedules, milestone completions, missed milestones, and the overlaps among partially concurrent activities is still a difficult and ambiguous undertaking.

One of the fundamental problems is the tendency of software projects to keep only the current values of schedule information, rather than a continuous record of events. For example, suppose the design of a project was scheduled to begin in January and end in June. In May it becomes apparent that June is unlikely, so July becomes the new target. In June, new requirements are levied, so the design stretches to August when it is nominally complete. Unfortunately, the original date (June in this example) and the intermediate date (July in this example) are often lost. Each time the plan of record is updated, the new date replaces the former date, which then disappears from view.

It would be very useful and highly desirable to keep track of each change to a schedule, why the schedule changed, and what were the prior schedules for completing the same event or milestone.

Another ambiguity of software measurement is the lack of any universal agreement as to what constitutes the major milestones of software projects. A large minority of projects tend to use "completion of requirements," "completion of design," "completion of coding," and "completion of testing" as major milestones. However, many other activities are on the critical path for completing software projects, and there may not be any formal milestones for these: examples include "completion of user documentation," and "completion of patent search."

It is widely known that those of us in software normally commence the next activity in a sequence long before the current activity is truly resolved and completed. Thus design usually starts before requirements are firm, coding starts before design is complete, and testing starts long before coding is completed. The classic "waterfall model" which assumes that activities flow from one to another in sequential order is actually a fiction which has almost never occurred in real life.

When an activity such as design starts before a predecessor activity such as requirements are finished, the author uses the term and metric called "overlap" to capture this phenomenon. For example, if the requirements for a project took four months, but design started on month three, then we would say that design overlapped requirements by 25%. As may be seen, overlap is defined as the amount of calendar time still remaining in an unfinished activity when a nominal successor activity begins.

The amount of overlap or concurrency among related activities should be a standard measurement practice, but in fact is often omitted or ignored. This is a truly critical omission, because there is no way to use historical data for accurate schedule prediction of future projects if this information is missing. (The average overlap on projects assessed by SPR is about 25%, but the range goes up to 100%.)

Simplistic total schedule measurements along the lines of, "The project started in August of 2006 and was finished in May of 2008 are essentially worthless for serious productivity studies. Accurate and effective schedule measurement would include the schedules of specific activities and tasks, and also include the network of concurrent and overlapped activities. Further, any changes to the original schedules would be recorded too.

Gaps and Errors in Measuring Creeping Requirements

One of the advantages of function point measurements, as opposed to "lines of code" metrics, is that function points are capable of measuring requirements growth. The initial function point analysis can be performed from the initial user requirements. Later when the application is delivered to users, a final function point analysis is performed.

By comparing the quantity of function points at the end of the requirements phase with the volume of function points of the delivered application, it has been found that the average rate of growth of creeping requirements is between 1% and 3% per calendar month, and sometimes more. The largest volume of creeping requirements noted by the author was an astonishing 289%.

Creeping requirements are one of the major reasons for cost overruns and schedule delays. For every requirements change of 10 function points about 10 calendar days will be added to project schedules and more than 20 hours of staff effort. It is important to measure the quantity of creeping requirements, and it is also important for contracts and outsource agreements to include explicit terms for how they will be funded.

It is also obvious that as requirements continue to grow and change, it will be necessary to factor in the changes and produce new cost estimates and new schedule plans. Yet in a majority of U.S. software projects creeping requirements are neither measured explicitly nor factored into project plans. This problem is so hazardous that it really should be viewed as professional malpractice on the part of management.

There are effective methods for controlling change but these are not always used. Joint Application Design (JAD), formal requirements inspections, and change control boards have long track records of success. Yet these methods are almost never used in applications that end up in court for cancellation or major overruns.

The Hazards and Errors of "Lines of Code" Metrics

One of the oldest and most widely used metrics for software has been that of "lines of code" which is usually abbreviated to "LOC." Unfortunately this metric is one of the most ambiguous and hazardous metrics in the history of business. LOC metrics are ambiguous because they can be counted using either physical lines or logical statements. LOC metrics are hazardous because they penalize high-level programming languages, and can't be used to measure non-coding activities.

The development of Visual Basic and its many competitors have changed the way many modern programs are developed. Although these visual languages do have a procedural source code portion, quite a bit of the more complex kinds of "programming" are done using button controls, pull-down menus, visual worksheets, and reusable components. In other words, programming is being done without anything that can be identified as a "line of code" for measurement or estimation purposes. By today in 2008 perhaps 60% of new software applications are developed using either object-oriented languages or visual languages (or both). Indeed, sometimes as many as 12 to 15 different languages are used in the same applications.

For large systems, programming itself is only the fourth most expensive activity. The three higher-cost activities cannot be measured or estimated effectively using the lines of code metric. Also, the fifth major cost element, project management, cannot easily be estimated or measured using the LOC metric either. Table 14 shows the rank order of software cost elements for large applications in descending order:

Table 14: Rank Order of Large System Software Cost Elements

1. Defect removal (inspections, testing, finding and fixing bugs)
2. Producing paper documents (plans, specifications, user manuals)
3. Meetings and communication (clients, team members, managers)
4. Programming
5. Project management

The usefulness of a metric such as lines of code which can only measure and estimate one out of the five major software cost elements of software projects is a significant barrier to economic understanding.

Following is an excerpt from the 3rd edition of the author's book Applied Software Measurement (McGraw Hill, 2008) that illustrates the economic fallacy of KLOC metrics:

"The reason that LOC metrics give erroneous results with high-level languages is because of a classic and well known business problem: the impact of fixed costs. Coding itself is only a small fraction of the total effort that goes into software. Paperwork in the form of plans, specifications, and user documents often cost much more. Paperwork tends to act

like a fixed cost, and that brings up a well-known rule of manufacturing: "*when a manufacturing process includes a high percentage of fixed costs and there is a reduction in the number of units manufactured, the cost per unit will go up*."

Here are two simple examples, showing both the lines-of-code results and function point results for doing the same application in two languages: basic assembly and C++. In Case 1 we will assume that an application is written in assembly. In Case 2 we will assume that the same application is written in C++.

Case 1: Application Written in the Assembly Language

Assume that the assembly language program required 10,000 lines of code, and the various paper documents (specifications, user documents, etc.) totaled to 100 pages. Assume that coding and testing required 10 months of effort, and writing the paper documents took 5 months of effort. The entire project totaled 15 months of effort, and so has a productivity rate of 666 LOC per month. At a cost of $10,000 per staff month the application cost $150,000. Expressed in terms of cost per source line, the costs are $15.00 per line of source code.

Case 2: The Same Application Written in the C++ Language

Assume that C++ version of the same application required only 1,000 lines of code. The design documents probably were smaller as a result of using an O-O language, but the user documents are the same size as the previous case: assume a total of 75 pages were produced. Assume that coding and testing required 1 month and document production took 4 months. Now we have a project where the total effort was only 5 months, but productivity expressed using LOC has dropped to only 200 LOC per month. At a cost of $10,000 per staff month the application cost $50,000 or only one third as much as the assembly language version. The C++ version is a full $100,000 cheaper than the assembly version, so clearly the C++ version has much better economics. But the cost per source line for this version has jumped to $50.00.

Even if we measure only coding, we still can't see the value of high-level languages by means of the LOC metric: the coding rates for both the assembly language and C++ versions were both identical at 1,000 LOC per month, even though the C++ version took only 1 month as opposed to 10 months for the assembly version.

Since both the assembly and C++ versions were identical in terms of features and functions, let us assume that both versions were 50 function points in size. When we express productivity in terms of function points per staff month, the assembly version had a productivity rate of 3.33 function points per staff month. The C++ version had a productivity rate of 10 function points per staff month. When we turn to costs, the assembly version had a cost of $3000 per function point while the C++ version had a cost of $1000 per function point. Thus function point metrics clearly match the assumptions of standard economics, which define productivity as "*goods or services produced per unit of labor or expense*."

Lines of code metrics, on the other hand, do not match the assumptions of standard economics and in fact show a reversal. Lines of code metrics distort the true economic case by so much that their use for economic studies involving more than one programming language might be classified as professional malpractice.

The only situation where LOC metrics behave reasonably well is when two projects utilize the same programming language. In that case, their relative productivity can be measured with LOC metrics. But if two or more different languages are used, the LOC results will be economically invalid."

A Short History of Lines of Code (LOC) Metrics

It is interesting to consider the history of LOC metrics and some of the problems with LOC metrics that led IBM to develop function point metrics. Following is a brief history from 1960 through today, with projections to 2010:

Circa 1960: When the LOC metric was first introduced there was only one programming language and that was basic assembly language. Programs were small and coding effort comprised about 90% of the total work. Physical lines and logical statements were the same thing for basic assembly. In this early environment, LOC metrics were useful for both economic and quality analysis. Unfortunately as the software industry changed, the LOC metric did not change and so became less and less useful until by about 1980 it had become extremely harmful without very many people realizing it.

Circa 1970: By 1970 basic assembly had been supplanted by macro-assembly. The first generation of higher-level programming languages such as COBOL, FORTRAN, and PL/I were starting to be used. The first known problem with LOC metrics was in 1970 when many IBM publication groups exceeded their budgets for that year. It was discovered (by the author) that technical publication group budgets had been based on 10% of the budget for programming. The publication projects based on assembly language did not overrun their budgets, but manuals for the projects coded in PL/S (a derivative of PL/I) had major overruns. This was because PL/S reduced coding effort by half, but the technical manuals were as big as ever. The initial solution was to give a formal mathematical definition to language "levels." The "level" was the number of statements in basic assembly language needed to equal the functionality of 1 statement in a higher-level language. Thus COBOL was a "level 3" language because it took 3 assembly statements to equal 1 COBOL statement. Using the same rule, SMALLTALK is a level 18 language. The documentation problem was one of the reasons IBM assigned Allan Albrecht and his colleagues to develop function point metrics. Also macro assembly language had introduced reuse, and had also begun the troublesome distinction between physical lines of code and logical statements. The percentage of project effort devoted to coding was dropping from 90% down to about 50%, and LOC metrics were no longer effective for economic or quality studies. After function point metrics were developed circa 1975 the definition of "language level" was expanded to include the number of logical code statements equivalent to 1 function point. COBOL, for example

requires about 105 statements per function point in the procedure and data divisions. This expansion is the mathematical basis for "backfiring" or direct conversion from source code to function points. Of course individual programming styles make backfiring a method with poor accuracy.

Circa 1980: By about 1980 the number of programming languages had topped 50 and object-oriented languages were rapidly evolving. As a result, software reusability was increasing rapidly. Another issue circa 1980 was the fact that many applications were starting to use more than one programming language, such as COBOL and SQL. In the middle of this decade the first commercial software cost estimating tool based on function points had reached the market, SPQR/20. By the end of this decade coding effort was below 35% of total effort, and LOC was no longer valid for either economic or quality studies. LOC metrics could not quantify requirements and design defects, which now outnumbered coding defects. LOC metrics could not be used to measure any of the non-coding activities such as requirements, design, documentation, or project management. The response of the LOC users to these problems was unfortunate: they merely stopped measuring anything but code production and coding defects. The bulk of all published reports based on LOC metrics cover less than 35% of development effort and less than 25% of defects, with almost no data being published on requirements and design defects, rates of requirements creep, design costs, and other modern problems.

Circa 1990: By about 1990 not only were there more than 500 programming languages in use, but some applications were written in 12 to 15 different languages. There were no international standards for counting code, and many variations were used sometimes without being defined. A survey of software journals in 1993 found that about one third of published articles used physical lines, one third used logical statements, and the remaining third used LOC metrics without even bothering to say how they were counted. Since there is about a 500% variance between physical LOC and logical statements for many languages, this was not a good situation. Even worse the arrival of Visual Basic introduced a class of programming language where counting lines of code was not possible. This is because a lot of Visual Basic "programming" was not done with procedural code but rather with buttons and pull-down menus. In the middle of this decade a controlled study was done that used both LOC metrics and function points for 10 versions of the same application written in 10 different programming languages including four object-oriented languages. This study was published in American Programmer in 1994. This study found that LOC metrics violated the basic concepts of economic productivity and penalized high-level and OO languages due to the fixed costs of requirements, design, and other non-coding activities. This was the first published study to state that LOC metrics constituted professional malpractice if used for economic studies where more than one programming language was involved. By the 1990'a most consulting studies that collected benchmark and baseline data used function points. There are no large-scale benchmarks based on LOC metrics. The International Software Benchmark Standards Group (ISBSG) was formed in 1997 and only publishes data in function point form. By the end of the decade, some projects were spending less than 20% of the total effort on coding, so LOC metrics could not be used for the 80% of effort outside the coding domain. The LOC users remained blindly indifferent to these

problems, and continued to measure only coding, while ignoring the overall economics of complete development cycles that include requirements, analysis, design, user documentation, project management, and many other non-coding tasks. By the end of the decade non-coding defects in requirements and design outnumbered coding defects almost 2 to 1. But since non-code defects could not be measured with LOC metrics the LOC literature simply ignores them.

Circa 2000: By the end of the century the number of programming languages had topped 700 and continues to grow at more than 1 new programming language per month. Web applications are mushrooming, and all of these are based on very high-level programming languages and substantial reuse. The Agile methods are also mushrooming, and also tend to use high-level programming languages. Software reuse in some applications now tops 80%. LOC metrics cannot be used for most web applications and are certainly not useful for measuring Scrum sessions and other non-coding activities that are part of Agile projects. Function point metrics have become the dominant metric for serious economic and quality studies. But two new problems have appeared that have kept function point metrics from actually becoming the industry standard for both economic and quality studies. The first problem is the fact that some software applications are now so large (>300,000 function points) that normal function point analysis is too slow and too expensive to be used. The second problem is that the success of function points has triggered an explosion of function point "clones." As of 2008 there are at least 24 function point variations. This makes benchmark and baseline studies difficult, because there are very few conversion rules from one variation to another. Although LOC metrics continue to be used, they continue to have such major errors that they constitute professional malpractice for economic and quality studies where more than one language is involved, or where non-coding issues are significant.

Circa 2010: It would be nice to predict an optimistic future, but if current trends continue within a few more years the software industry will have more than 800 programming languages of which about 750 will be obsolete or becoming dead languages, more than 20 variations for counting lines of code, more than 50 variations for counting function points, and probably another 20 unreliable metrics such as "cost per defect" or percentages of unstable numbers. Future generations of sociologists will no doubt be interested in why the software industry spends so much energy on creating variations of things, and so little energy on fundamental issues. No doubt large projects will still be cancelled, litigation for failures will still be common, software quality will still be bad, software productivity will remain low, security flaws will be alarming, and the software literature will continue to offer unsupported claims without actually presenting quantified data. What the software industry needs is actually fairly straightforward: 1) measures of defect potentials from all sources expressed in terms of function points; 2) measures of defect removal efficiency levels for all forms of inspection and testing; 3) activity-based productivity benchmarks from requirements through delivery and then for maintenance and customer support from delivery to retirement using function points; 4) certified sources of reusable material near the zero-defect level; 5) much improved security methods to guard against viruses, spyware, and hacking; 6) licenses and board-certification for software engineering specialties. But

until measurement becomes both accurate and cost-effective, none of these are likely to occur. An occupation that will not measure its own performance with accuracy is not a true profession.

The Hazards and Errors of the "Cost per Defect" Metric

The well-known and widely cited "cost per defect measure" also violates the canons of standard economics. Although this metric is often used to make quality 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.

Because of the way cost per defect is normally measured, as quality improves, "cost per defect" steadily increases until zero-defect software is achieved, at which point the metric cannot be used at all.

As with KLOC metrics, 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 $62.50 per hour. Assume that all three software features that are being tested are 100 function points in size.

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.

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 $1,875. As quality improves, "cost per defect" rises sharply. Let us know consider cost per function point. 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 slack 15 hours of slack time are backed out, leaving 25 hours for actual testing, the costs would have been $1,562. With slack time removed, the cost per function point would be $15.63. 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 4 hours to repair. There are some bugs that are expensive and 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.

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. It is because the costs of writing test cases, running them, and having maintenance personnel available act as fixed costs. In any manufacturing cycle with a high percentage of fixed costs, the cost per unit will go up as the number of units goes down. This basic fact of manufacturing economics is why both "cost per defect" and "lines of code" are hazardous and invalid for economic analysis of software applications.

The Hazards of Multiple Metrics without Conversion Rules

There are many sciences and engineering disciplines that have multiple metrics for the same values. For example we have nautical miles, statute miles, and kilometers for measuring speed and distance. We have Fahrenheit and Celsius for measuring temperature. We have three methods for measuring the octane ratings of gasoline. However, other engineering disciplines have conversion rules from one metric to another.

The software industry is unique in having more metric variants than any other engineering discipline in history, combined with an almost total lack of conversion rules from one metric to another. As a result, producing accurate benchmarks of software productivity and quality is much harder than for any other engineering field.

The author has identified five distinct variations in methods for counting lines of code, and 24 distinct variations in counting function point metrics. There are no standard conversion rules between any of these variants.

Here is an example of why this situation is harmful to the industry. Suppose you are a consultant who has been commissioned by a client to find data on the costs and schedules of producing a certain kind of software, such as a PBX switching system. You scan the literature and benchmark data bases and discover that data exists on 66 similar projects. You would like to perform a statistical analysis of the results for presentation to the client. But now the problems begin when trying to do statistical analysis of the 66 samples:

1. Three were measured using lines of code, and counted physical lines.
2. Three were measured using lines of code, and counted logical statements.
3. Three were measured using lines of code, and did not state the counting method.
4. Three were constructed from reusable objects and only counted custom code
5. Three were measured using IFPUG function point metrics.
6. Three were measured using COSMIC function point metrics.
7. Three were measured using Full function points.
8. Three were measured using Mark II function point metrics.
9. Three were measured using FESMA function points
10. Three were measured using NESMA function points.
11. Three were measured using unadjusted function points.
12. Three were measured using Engineering function points.
13. Three were measured using Web-object points.
14. Three were measured using Function points light.
15. Three were measured using backfire function point metrics.
16. Three were measured using Feature points.
17. Three were measured using Story points.
18. Three were measured using Use Case points.
19. Three were measured using MOOSE metrics.
20. Three were measured using goal-question metrics.
21. Three were measured using TSP/PSP task hours.
22. Three were measured using RTF metrics.

As of 2009 there are no effective conversion rules between any of these metrics. There is no effective way of performing a statistical analysis of results. Why the software industry has developed so many competing variants of software metrics is an unanswered sociological question.

Developers of new versions of function point metrics almost always fail to provide conversion rules between their new version and older standard metrics such as IFPUG function points. In the author's view it is the responsibility of the developers of new metrics to provide conversion rules to older metrics. It is not the responsibility of organization such as IFPUG to provide conversion rules to scores of minor variations in counting practices.

The existence of five separate methods for counting source code and at least 24 variations in counting function points with almost no conversion rules from one metric to another is a professional embarrassment to the software industry. As of 2009 the plethora of ambiguous metrics is slowing progress towards a true economic understanding of the software industry.

However a technical solution to this problem does exist. By using the high-speed pattern-matching method for function point size prediction in Software Risk Master™, it would be possible to perform separate size estimates for each of the 22 examples shown above. The high-speed method embedded in the Software Risk Master™ tool produces size in terms of both IFPUG function points and logical code statements. This tool requires only between 1 and 5 minutes per application for sizing applications between 1 and 300,000 function points.. The tool can also be used from before requirements start through development and also for legacy applications. It can also be used on commercial packages, open-source applications, and even classified applications if they can be placed on a standard taxonomy of nature, scope, class, type, and complexity.

Not only would this tool provide size in terms of standard IFPUG function points, but the taxonomy that is included with the tool would facilitate large-scale benchmark studies. After samples of perhaps 100 applications were sized which had used story points, use-case points, or task hours enough data would become available to perform useful statistical studies of the size ratios of all common metrics.

Gaps and Errors in Measuring Software Quality

Historically the costs of finding and fixing bugs outweigh every other cost element for large software projects. Given the fact that defect repairs are the major source of software cost overruns and also schedule slippage, one might think that defect measurements would be accurate and complete. Unfortunately quality measurement is one of the weakest links in the entire chain of software measurements.

Technically it is fairly easy to measure software quality, but a majority of companies and government organizations fail to do so. In fact about 50% of U.S. companies don't have any quality measurements at all. Many of the companies that have rudimentary quality measures don't even begin to measure bugs until customers start to report them after delivery of the applications.

Only about 10% of the most sophisticated companies have full-lifecycle quality measurements that include measures of defect potentials, defect removal efficiency for both inspections and testing, bad-fix injections, and other key quality measures.

The measured range of defect potentials ranges from just below 2.00 defects per function point to about 10.00 defects per function point. Defect potentials correlate with application size. As application sizes increase, defect potentials also rise. (Defect

potentials also vary with the type of software, with CMM levels, and in response to other factors.)

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

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

As of 2008 the U.S. averages for defect removal efficiency against each of the five major defect categories is approximately the following:

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

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

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

Latent defects found after the 90 day period can exist for years, but on average about 50% of residual latent defects are found each calendar year. The results vary with number of users of the applications. The more users, the faster residual latent defects are discovered. Results also vary with the nature of the software itself. Military, embedded, and systems software tends to find bugs or defect more quickly than information systems.) Table 15 shows defect potentials for various kinds of software

Table 15: Average Defect Potentials for Six Application Types (Data expressed in "defects per function point"

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

Note that IFPUG function points, version 4.2 are used in this paper for expressing results. The form of defect called "bad fix" in table 1 is that of secondary defects accidentally present in a bug or defect repair itself.

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

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

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

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

Measuring Defect Removal Efficiency

Most forms of testing average only about 30% to 35% in defect removal efficiency levels and seldom top 50%. Formal design and code inspections, on the other hand, often top

85% in defect removal efficiency and average about 65%. With every form of defect removal having a comparatively low level of removal efficiency, it is obvious that many separate forms of defect removal need to be carried out in sequence to achieve a high level of cumulative defect removal. The phrase "cumulative defect removal" refers to the total number of defects found before the software is delivered to its customers.

Table 16 shows patterns of defect prevention and defect removal for the same six forms of software shown in Table 15:

Table 16: Patterns of Defect Prevention and Removal Activities

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

Some forms of defect removal such as desk checking and unit testing are normally performed privately by individuals and are not usually measured. However several companies such as IBM and ITT have collected data on these methods from volunteers, who agreed to record "private" defect removal activities in order to judge their relative

defect removal efficiency levels. (The author himself was once such a volunteer, and was troubled to note that his private defect removal activities were less than 35% efficient.)

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

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

As can be seen from the short discussions here, measuring defect potentials and defect removal efficiency provide the most effective known ways of evaluating various aspects of software quality control.

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

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

When IBM applied formal inspections to a large data base project, it was interesting that delivered defects were reduced by more than 50% from previous releases. The overall schedule was shortened by about 15%. Testing itself was reduced from two shifts over a 60 day period to one shift over a 40 day period. More important, customer satisfaction improved to "good" from prior releases, where customer satisfaction had been "very

poor". Cumulative defect removal efficiency was raised from about 80% to just over 95% as a result of using formal design and code inspections.

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

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

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

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

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

Gaps in Measuring Tools, Methodologies, and Programming Languages

As this paper has pointed out, collecting accurate quantified data about software effort, schedules, staffing, costs, and quality is seldom done well and often not done at all. But even if accurate quantitative data were collected the data by itself would not be sufficient to explain the variations that occur in project outcomes.

In addition to quantitative data, it is also necessary to record a wide variety of supplemental topics in order to explain the variations that occur. Examples of the kinds of supplemental data include:

1. The level of the development team on the CMM or CMMI scale
2. Methodologies were used such as Agile, Rational Unified Process (RUP), etc.
3. Whether or not Six-Sigma analysis was utilized
4. Whether or not Quality Function Deployment (QFD) was utilized
5. The kinds of project management and estimating tools used
6. What combination of 700 programming languages were utilized
7. Whether the project utilized subcontractors or off-shore developers
8. The specific combination of inspections and tests were used
9. Whether automated test tools were utilized
10. Whether Joint Application Design (JAD) was utilized
11. Whether prototypes were built
12. The kinds of change control methods utilized

This kind of information lacks any kind of standard representation. The author's approach uses multiple-choice questions to ascertain the overall pattern of tool and methodology usage. At the end of the questionnaire, space is provided to name the specific tools, languages, and other factors that had a significant effect on the project.

There are a number of widely used questionnaires that gather supporting data on methods, tools, languages, and other factors that influence software outcomes. The oldest of these is perhaps the Software Productivity Research (SPR) questionnaire which was first used in 1983. The assessment questionnaire created by the Software Engineering Institute (SEI) is perhaps the second, and became available about 1985.

Since then dozens of consulting and benchmark organizations have created questionnaires for collecting data about software tools and methods. Some of these organizations, in alphabetical order, include the David's Consulting Group, Galorath Associates, Gartner Group, the International Software Benchmark Standards Group (ISBSG), and Quantitative Software Management (QSM). There are many more in addition to these.

Each of these questionnaires may be useful in its own right, but because they are all somewhat different and there are no industry standards that define the information to be collected, it is hard to carry out large-scale studies.

Applying Standard Economics to Software Applications

Because software has such high labor content and such dismal quality control, it is of considerable importance to be able to measure productivity and quality using standard economic principles. It is also of considerable importance to be able to predict productivity and quality before major projects start. In order to accomplish these basic goals, a number of standards need to be adopted for various measurement topics. These standards include:

1. A standard taxonomy for identifying projects without ambiguity.
2. A set of standard charts of accounts for collecting activity-level data.

3. A standard for measuring schedule slippage.
4. A standard for measuring requirements creep and requirements churn.
5. A set of standard charts of accounts for maintenance (defect repairs).
6. A set of standard charts of accounts for enhancements (new features).
7. A set of standard charts of accounts for customer support and service.
8. A standard questionnaire for collecting methodology and tool information.
9. A standard definition for when projects start.
10. A standard definition for when projects end.
11. A standard definition for defect origins and defect potentials.
12. A standard definition for defect removal efficiency.
13. A standard checklist for measuring the contribution of specialists.
14. Standard job descriptions for software specialists.
15. Multiple standard conversion rules between unlike measurements.
16. A standard for applying "cost of quality" to software projects.

There are more issues than the 16 shown here, but until these 16 are addressed and their problems solved, it will not be possible to use standard economic assumptions for software applications. What the author suggests would be a continuing series of workshops that involved the major organizations that are concerned with software: the SEI, IFPUG, COSMIC, ISBSG, ITMPI, PMI, ISO, etc. Universities should also be involved. Unfortunately as of 2008 the relationships among these organizations tends to be somewhat adversarial. Each wants its own method to become the basis of international standards. Therefore cooperation on common problems is difficult to bring about.

SUMMARY AND CONCLUSIONS

Software is one of the driving forces of modern industry and government operations. Software controls almost every complicated device now used by human beings. But software remains a difficult and intractable discipline that is hard to predict and hard to measure. The quality of software is embarrassingly bad. Given the economic importance of software, it is urgent to make software development and maintenance true engineering disciplines, as opposed to art forms or skilled crafts.

In order to make software engineering a true engineering discipline and a true profession, much better measurement practices are needed than have been utilized to date. Quantitative data and qualitative data need to be collected in standard fashions that are amenable to statistical analysis.

SUGGESTED READINGS ON SOFTWARE MEASURES AND ISSUES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Jones, Capers; Conflict and Litigation Between Software Clients and Developers; Software Productivity Research, Inc.; Narragansett, RI; 2008; 45 pages.

Jones, Capers; "Preventing Software Failure: Problems Noted in Breach of Contract Litigation"; Capers Jones & Associates, Narragansett, RI; 2008; 25 pages.

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

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

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

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

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

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

Web Sites

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

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

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

Project Management Institute (www.PMI.org)