ACHIEVING SOFTWARE EXCELLENCE

Version 8.0 – October 4, 2016

Abstract

As of the year 2016 software applications are the main operational component of every major business and government organization in the world. But software quality is still not good for a majority of these applications. Software schedules and costs are both frequently much larger than planned. Cyber-attacks are become more frequent and more serious.

This study discusses the proven methods and results for achieving software excellence. The paper also provides quantification of what the term "excellence" means for both quality and productivity. Formal sizing and estimating using parametric estimation tools, excellent progress and quality tracking also using special tools, and a comprehensive software quality program can lead to shorter schedules, lower costs, and higher quality at the same time.

Capers Jones, VP and CTO, Namcook Analytics LLC

Email: Capers.Jones3@gmail.com

Web: www.Namcook.com

INTRODUCTION

Software is the main operating tool of business and government in 2016. But software quality remains marginal; software schedules and costs remained much larger than desirable or planned. Cancelled projects are about 35% in the 10,000 function point size range and about 5% of software outsource agreements end up in court in litigation. Cyber-attacks are increasing in numbers and severity. This short study identifies the major methods for bringing software under control and achieving excellent results.

The first topic of importance is to show the quantitative differences between excellent, average, and poor software projects in quantified form. Table 1 shows the essential differences between software excellence, average, and unacceptably poor results for a mid-sized project of 1,000 function points or about 53,000 Java statements.

The data comes from benchmarks performed by Namcook Analytics LLC. These were covered by non-disclosure agreements so specific companies are not shown. However the "excellent" column came from technology and medical device companies; the average from insurance and manufacturing; and the poor column from state and local governments:

Table 1: Comparisons of Excellent, Average, and Poor Software Results

Topics	**Excellent**	**Average**	**Poor**
Monthly Costs			
(Salary + overhead)	$10,000	$10,000	$10,000
Size at Delivery			
Size in function points	1,000	1,000	1,000
Programming language	Java	Java	Java
Language Levels	6.25	6.00	5.75
Source statements per funct. point	51.20	53.33	55.65
Size in logical code statements	51,200	53,333	55,652
Size in KLOC	51.20	53.33	55.65
Certified reuse percent	20.00%	10.00%	5.00%
Quality			
Defect potentials	**2,818**	**3,467**	**4,266**
Defects per function point	2.82	3.47	4.27
Defects per KLOC	55.05	65.01	76.65
Defect removal efficiency (DRE)	**99.00%**	**90.00%**	**83.00%**
Delivered defects	**28**	**347**	**725**
High-severity defects	**4**	**59**	**145**

Security vulnerabilities	**2**	**31**	**88**
Delivered per function point	0.03	0.35	0.73
Delivered per KLOC	0.55	6.50	13.03
Key Quality Control Methods			
Formal estimates of defects	**Yes**	**No**	**No**
Formal inspections of deliverables	**Yes**	**No**	**No**
Static analysis of all code	**Yes**	**Yes**	**No**
Formal test case design	**Yes**	**Yes**	**No**
Testing by certified test personnel	**Yes**	No	**No**
Mathematical test case design	**Yes**	No	**No**
Project Parameter Results			
Schedule in calendar months	12.02	13.80	18.20
Technical staff + management	6.25	6.67	7.69
Effort in staff months	75.14	92.03	139.98
Effort in staff hours	9,919	12,147	18,477
Costs in Dollars	**$751,415**	**$920,256**	**$1,399,770**
Cost per function point	$751.42	$920.26	$1,399.77
Cost per KLOC	$14,676	$17,255	$25,152
Productivity Rates			
Function points per staff month	13.31	10.87	7.14
Work hours per function point	9.92	12.15	18.48
Lines of code per staff month	681	580	398
Cost Drivers			
Bug repairs	**25.00%**	**40.00%**	**45.00%**
Paper documents	20.00%	17.00%	20.00%
Code development	35.00%	18.00%	13.00%
Meetings	8.00%	13.00%	10.00%
Management	12.00%	12.00%	12.00%
Total	100.00%	100.00%	100.00%
Methods, Tools, Practices			
Development Methods	TSP/PSP	Agile	Waterfall
Requirements Methods	JAD	Embedded	Interview
CMMI Levels	5	3	1
Work hours per month	132	132	132
Unpaid overtime	0	0	0
Team experience	Experienced	Average	Inexperienced
Formal risk analysis	**Yes**	**Yes**	**No**

Formal quality analysis	**Yes**	**No**	**No**
Formal change control	**Yes**	**Yes**	**No**
Formal sizing of project	**Yes**	**Yes**	**No**
Formal reuse analysis	**Yes**	**No**	**No**
Parametric estimation tools	**Yes**	**No**	**No**
Inspections of key materials	**Yes**	**No**	**No**
Static analysis of all code	**Yes**	**Yes**	**No**
Formal test case design	**Yes**	**No**	**No**
Certified test personnel	**Yes**	**No**	**No**
Accurate status reporting	**Yes**	**Yes**	**No**
Accurate defect tracking	**Yes**	**No**	**No**
More than 15% certified reuse	**Yes**	**Maybe**	**No**
Low cyclomatic complexity	**Yes**	**Maybe**	**No**
Test coverage > 95%	**Yes**	**Maybe**	**No**

As stated the data in table 1 comes from the author's clients, which consist of about 750 companies of whom 150 are Fortune 500 companies. About 40 government and military organizations are also clients, but the good and average columns in table 1 are based on corporate results rather than government results. State and local governments provided data for the poor quality column.

(Federal Government and defense software tend to have large overhead costs and extensive status reporting that are not found in the civilian sector. Some big defense projects have produced so much paperwork that there were over 1,400 English words for every Ada statement, and the words cost more than the source code.)

(*Note that the data in this report was produced using the Namcook Analytics Software Risk Master™ (SRM) tool. SRM can operate as an estimating tool prior to requirements or as a benchmark measurement tool after deployment.)*

At this point it is useful to discuss and explain the main differences between the best, average, and poor results.

Software Sizing, Estimating, and Project Tracking Differences

High-quality projects with excellent results all use formal parametric estimating tools, perform formal sizing before starting, and have accurate status and cost tracking during development.

A comparative study by the author of accuracy differences between manual estimates and parametric estimates showed that the manual estimates averaged about 34% optimistic for schedules and costs.

Worse, manual estimating errors increased with application size. Below 250 function points manual and parametric estimates were both within 5%. Above 10,000 function points manual estimates were optimistic by almost 40% while parametric estimates were often within 10%. Overall parametric estimates usually differed by less than 10% from actual results for schedules and costs, sometimes less than 5%, and were almost never optimistic.

The parametric estimation tools included COCOMO, Excelerator, KnowledgePlan, SEER, SLIM, Software Risk Master, and TruePrice. All of these parametric tools were more accurate than manual cost and schedule estimates for all size ranges and application types.

High-quality projects also track results with high accuracy for progress, schedules, defects, and cost accumulation. Some excellent projects use specialized tracking tools such as Computer Aid's Automated Project Office (APO) which was built to track software projects. Others use general tools such as Microsoft Project which supports many kinds of projects in addition to software.

Average projects with average results sometimes used parametric estimates but more often use manual estimates. However some of the average projects did utilize estimating specialists, who are more accurate than untrained project managers.

Project tracking for average projects tends to be informal and use general-purpose tools such as Excel rather than specialized software tracking tools such as APO, Jira, Asana and others. Average tracking also "leaks" and tends to omit topics such as unpaid overtime and project management.

Poor quality projects almost always use manual estimates. Tracking of progress is so bad that problems are sometimes concealed rather than revealed. Poor quality cost tracking has major gaps and omits over 50% of total project costs. The most common omissions are unpaid overtime, project managers, and the work of part-time specialists such as business analysts, technical writers, and software quality assurance.

Quality tracking is embarrassingly bad and omits all bugs found before testing via static analysis or reviews, and usually omits bugs found during unit testing. Some poor-quality companies and government organizations don't track quality at all. Many others don't

track until late testing or deployment.

Software Quality Differences for Best, Average, and Poor Projects

Software quality is the major point of differentiation between excellent results, average results, and poor results.

While software executives demand high productivity and short schedules, the vast majority do not understand how to achieve them. Bypassing quality control does not speed projects up: it slows them down.

The number one reason for enormous schedule slips noted in breach of contract litigation where the author has been an expert witness is starting testing with so many bugs that test schedules are at least double their planned duration.

The major point of this article is: *High quality using a synergistic combination of defect prevention, pre-test inspections and static analysis combined with formal testing is fast and cheap*.

Poor quality is expensive, slow, and unfortunately far too common. Because most companies do not know how to achieve high quality, poor quality is the norm and at least twice as common as high quality.

High quality does not come from testing alone. It requires defect prevention such as Joint Application Design (JAD), quality function deployment (QFD) or embedded users; pre-test inspections and static analysis; and of course formal test case development combined with certified test personnel. New methods of test case development based on cause-effect graphs and design of experiments are quite a step forward.

The defect potential information in table 1 includes defects from five origins: requirements defects, design defects, code defects, document defects, and "bad fixes" or new defects accidentally included in defect repairs. The approximate distribution among these five sources is:

1.	Requirements defects	15%
2.	Design defects	30%
3.	Code defects	40%
4.	Document defects	8%
5.	Bad fixes	7%
6.	Total Defects	100%

Note that a "bad fix" is a bug in a bug repair. These can sometimes top 25% of bug repairs for modules with high cyclomatic complexity.

However the distribution of defect origins varies widely based on the novelty of the application, the experience of the clients and the development team, the methodologies

used, and programming languages. Certified reusable material also has an impact on software defect volumes and origins.

Table 2 shows approximate U.S. ranges for defect potentials based on a sample of 1,500 software projects that include systems software, web projects, embedded software, and information technology projects that range from 100 to 100,000 function points:

Table 2: Defect Potentials for 1,000 Projects

Defect Potentials	Projects	Percent
< 1.00	**5**	**0.50%**
2 to 1	**35**	**3.50%**
3 to 2	120	12.00%
4 to 3	425	42.50%
5 to 4	**350**	**35.00%**
> 5.00	**65**	**6.50%**
Totals	**1,000**	**100.00%**

It is unfortunate that buggy software projects outnumber low-defect projects by a considerable margin.

Because the costs of finding and fixing bugs have been the #1 cost driver for the entire software industry for more than 50 years, the most important difference between excellent and mediocre results are in the areas of defect prevention, pre-test defect removal, and testing.

All three examples are assumed to use the same set of test stages, including:

1. Unit test
2. Function test
3. Regression test
4. Component test
5. Performance test
6. System test
7. Acceptance test

The overall defect removal efficiency (DRE) levels of these 7 test stages range from

below 80% for the worst case up to about 95% for the best case.

Note that the seven test stages shown above are generic and used on a majority of software applications. Additional forms of testing may also be used, and can be added to SRM for specific clients and specific projects:

1. Independent testing (mainly government and military software)
2. Usability testing (mainly software with complex user controls)
3. Performance testing (mainly real-time software)
4. Security testing
5. Limits testing
6. Supply-chain testing
7. Nationalization testing (for international projects)

Testing alone is not sufficient to top 95% in defect removal efficiency (DRE). Pre-test inspections and static analysis are needed to approach or exceed the 99% range of the best case. Also requirements models and "quality-strong" development methods such as team software process (TSP) need to be part of the quality equation.

Excellent quality control

Excellent projects have rigorous quality control methods that include formal estimation of quality before starting, full defect measurement and tracking during development, and a full suite of defect prevention, pre-test removal and test stages. The combination of low defect potentials and high defect removal efficiency (DRE) is what software excellence is all about.

The most common companies that are excellent in quality control are usually the companies that build complex physical devices such as computers, aircraft, embedded engine components, medical devices, and telephone switching systems. Without excellence in quality these physical devices will not operate successfully. Worse, failure can lead to litigation and even criminal charges. Therefore all companies that use software to control complex physical machinery tend to be excellent in software quality.

Examples of organizations noted as excellent software quality in alphabetical order include Advanced Bionics, Apple, AT&T, Boeing, Ford for engine controls, General Electric for jet engines, Hewlett Packard for embedded software, IBM for systems software, Motorola for electronics, NASA for space controls, the Navy for surface weapons, Raytheon, and Siemens.

Companies and projects with excellent quality control tend to have low levels of code cyclomatic complexity and high test coverage; i.e. test cases cover > 95% of paths and risk areas.

These companies also measure quality well and all know their defect removal efficiency

(DRE) levels. (Any company that does not measure and know their DRE is probably below 85% in DRE.)

Excellent quality control has defect removal efficiency levels (DRE) between about 97% for large systems in the 10,000 function point size range and about 99.6% for small projects < 1,000 function points in size.

A DRE of 100% is theoretically possible but is extremely rare. The author has only noted DRE of 100% in 10 projects out of a total of about 25,000 projects examined. As it happens the projects with 100% DRE were all compilers and assemblers built by IBM and using > 85% certified reusable materials. The teams were all experts in compilation technology and of course a full suite of pre-test defect removal and test stages were used as well.

Average quality control

In today's world agile is the new average. Agile development has proven to be effective for smaller applications below 1,000 function points in size. Agile does not scale up well and is not a top method for quality. Agile is weak in quality measurements and does not normally use inspections, which have the highest defect removal efficiency (DRE) of any known form of defect removal. Disciplined Agile Development (DAD) can be used successfully on large systems where vanilla agile/scrum is not effective. Inspections top 85% in DRE and also raise testing DRE levels. Among the author's clients that use Agile the average value for defect removal efficiency is about 92% to 94%. This is certainly better than the 85% to 90% industry average for waterfall projects, but not up to the 99% actually needed to achieve optimal results.

Some but not all agile projects use "pair programming" in which two programmers share an office and a work station and take turns coding while the other watches and "navigates." Pair programming is very expensive but only benefits quality by about 15% compared to single programmers. Pair programming is much less effective in finding bugs than formal inspections, which usually bring 3 to 5 personnel together to seek out bugs using formal methods.

Agile is a definite improvement for quality compared to waterfall development, but is not as effective as the quality-strong methods of team software process (TSP) and the rational unified process (RUP) for larger applications > 1000 function points. An average agile project among the author's clients is about 275 function points. Disciplined agile development (DAD) is a good choice for larger information software applications.

Average projects usually do not know defects by origin, and do not measure defect removal efficiency until testing starts; i.e. requirements and design defects are under reported and sometimes invisible.

A recent advance in software quality control now frequently used by average as well as advanced organizations is that of static analysis. Static analysis tools can find about 55%

of code defects, which is much higher than most forms of testing.

Many test stages such as unit test, function test, regression test, etc. are only about 35% efficient in finding code bugs, or find one bug out of three. This explains why 6 to 10 separate kinds of testing are needed.

The kinds of companies and projects that are "average" would include internal software built by hundreds of banks, insurance companies, retail and wholesale companies, and many government agencies at federal, state, and municipal levels.

Average quality control has defect removal efficiency levels (DRE) from about 85% for large systems up to 97% for small and simple projects.

Poor Quality Control

Poor quality control is characterized by weak defect prevention and almost a total omission of pre-test defect removal methods such as static analysis and formal inspections. Poor quality control is also characterized by inept and inaccurate quality measures which ignore front-end defects in requirements and design. There are also gaps in measuring code defects. For example most companies with poor quality control have no idea how many test cases might be needed or how efficient various kinds of test stages are.

Companies or government groups with poor quality control also fail to perform any kind of up-front quality predictions so they jump into development without a clue as to how many bugs are likely to occur and what are the best methods for preventing or removing these bugs.

One of the main reasons for the long schedules and high costs associated with poor quality is the fact that so many bugs are found when testing starts that the test interval stretches out to two or three times longer than planned.

Some of the kinds of software that are noted for poor quality control include the Obamacare web site, municipal software for property tax assessments, and software for programmed stock trading, which has caused several massive stock crashes.

Poor quality control is often below 85% in defect removal efficiency (DRE) levels. In fact for canceled projects or those that end up in litigation for poor quality, the DRE levels may drop below 80%, which is low enough to be considered professional malpractice. In litigation where the author has been an expert witness DRE levels in the low 80% range have been the unfortunate norm.

Table 3 shows the ranges in defect removal efficiency (DRE) noted from a sample of 1,000 software projects. The sample included systems and embedded software, web projects, cloud projects, information technology projects, and also defense and commercial packages.

Table 3: Distribution of DRE for 1,000 Projects

DRE	Projects	Percent
> 99.00%	10	1.00%
95%-99%	120	12.00%
90%-94%	250	25.00%
85%-89%	475	47.50%
80%-85%	**125**	**12.50%**
< 80.00%	**20**	**2.00%**
Totals	**1,000**	**100.00%**

As can be seen high DRE does not occur often. This is unfortunate because projects that are above 95.00% in DRE have shorter schedules and lower costs than projects below 85.00% in DRE. The software industry does not measure either quality or productivity well enough to know this.

However the most important economic fact about high quality is: ***projects > 97% in DRE have shorter schedules and lower costs than projects < 90% in DRE.*** This is because projects that are low in DRE have test schedules that are at least twice as long as projects with high DRE due to omission of pre-test inspections and static analysis!

Reuse of Certified Materials for Software Projects

So long as software applications are custom designed and coded by hand, software will remain a labor-intensive craft rather than a modern professional activity. Manual software development even with excellent methodologies cannot be much more than 15% better than average development due to the intrinsic limits in human performance and legal limits in the number of hours that can be worked without fatigue.

The best long-term strategy for achieving consistent excellence at high speed would be to eliminate manual design and coding in favor of construction from certified reusable components.

It is important to realize that software reuse encompasses many deliverables and not just source code. A full suite of reusable software components would include at least the following 10 items:

Reusable Software Artifacts Circa 2016

1. **R**eusable requirements
2. Reusable architecture
3. Reusable design
4. Reusable code
5. Reusable project plans and estimates
6. Reusable test plans
7. Reusable test scripts
8. Reusable test cases
9. Reusable user manuals
10. Reusable training materials

These materials need to be certified to near zero-defect levels of quality before reuse becomes safe and economically viable. Reusing buggy materials is harmful and expensive. This is why excellent quality control is the first stage in a successful reuse program.

The need for being close to zero defects and formal certification adds about 20% to the costs of constructing reusable artifacts, and about 30% to the schedules for construction. However using certified reusable materials subtracts over 80% from the costs of construction and can shorten schedules by more than 60%. The more times materials are reused the greater their cumulative economic value.

One caution to readers: reusable artifacts may be treated as taxable assets by the Internal Revenue Service. It is important to check this topic out with a tax attorney to be sure that formal corporate reuse programs will not encounter unpleasant tax consequences.

The three samples in table 1 showed only moderate reuse typical for the start of 2016:

Excellent project	> 25% certified reuse
Average project	+ - 10% certified reuse
Poor projects	< 5% certified reuse

In the future it is technically possible to make large increases in the volumes of reusable materials. By around 2025 we should be able to construct software applications with perhaps 85% certified reusable materials. In fact some "mashup" projects already achieve 85% reuse, but the reused materials are not certified and some may contain significant bugs and security flaws.

Table 4 shows the productivity impact of increasing volumes of certified reusable materials. Table 4 uses whole numbers and generic values to simplify the calculations:

Table 4: Productivity Gains from Software Reuse
(Assumes 1,000 function points and 53,300 LOC)

Reuse Percent	Months of staff effort	Function Points per month	Work hours per function point	Lines of Code per month	Project Costs
0.00%	100	10.00	13.20	533	$1,000,000
10.00%	90	11.11	11.88	592	$900,000
20.00%	80	12.50	10.56	666	$800,000
30.00%	70	14.29	9.24	761	$700,000
40.00%	60	16.67	7.92	888	$600,000
50.00%	50	20.00	6.60	1,066	$500,000
60.00%	40	25.00	5.28	1,333	$400,000
70.00%	30	33.33	3.96	1,777	$300,000
80.00%	20	50.00	2.64	2,665	$200,000
90.00%	10	100.00	1.32	5,330	$100,000
100.00%	1	1,000.00	0.13	53,300	$10,000

Software reuse from certified components instead of custom design and hand coding is the only known technique that can achieve order-of-magnitude improvements in software productivity. True excellence in software engineering must derive from replacing costly and error-prone manual work with construction from certified reusable components.

Because finding and fixing bugs is the major software cost driver, increasing volumes of high-quality certified materials can convert software from an error-prone manual craft into a very professional high-technology profession. Table 3 shows probable quality gains from increasing volumes of software reuse:

Table5: Quality Gains from Software Reuse
(Assumes 1,000 function points and 53,300 LOC)

Reuse Percent	Defects per Function Point	Defect Potential	Defect Removal Efficiency	Delivered Defects
0.00%	5.00	1,000	90.00%	100
10.00%	4.50	900	91.00%	81
20.00%	4.00	800	92.00%	64
30.00%	3.50	700	93.00%	49
40.00%	3.00	600	94.00%	36
50.00%	2.50	500	95.00%	25
60.00%	2.00	400	96.00%	16
70.00%	1.50	300	97.00%	9
80.00%	1.00	200	98.00%	4
90.00%	0.50	100	99.00%	1
100.00%	-	1	99.99%	0

Since the current maximum for software reuse from certified components is only in the range of 15% or a bit higher, it can be seen that there is a large potential for future improvement.

Note that uncertified reuse in the form of mashups or extracting materials from legacy applications may top 50%. However uncertified reusable materials often have latent bugs, security flaws, and even error-prone modules so this not a very safe practices. In several cases the reused material was so buggy it had to be discarded and replaced by custom development.

Several emerging development methodologies such as “mashups” are pushing reuse values up above 90%. However the numbers and kinds of applications built from these emerging methods are small. Reuse needs to become generally available with catalogs of standard reusable components organized by industries: i.e. banking, insurance, telecommunications, firmware, etc.

Software Methodologies

Unfortunately selecting a methodology is more like joining a cult than making an informed technical decision. Most companies don't actually perform any kind of due diligence on methodologies and merely select the one that is most popular.

In today's world agile is definitely the most popular. Fortunately agile is also a pretty good methodology and much superior to the older waterfall method. However there are some caveats about methodologies.

Agile has been successful primarily for smaller applications < 1,000 function points in size. It has also been successful for internal applications where users can participate or be "embedded" with the development team to work our requirements issues.

Agile has not scaled up well to large systems > 10,000 function points. Agile has also not been visibly successful for commercial or embedded applications where there are millions of users and none of them work for the company building the software so their requirements have to be collected using focus groups or special marketing studies.

A variant of agile that uses "pair programming" or two programmers working in the same cubical with one coding and the other "navigating" has become popular. However it is very expensive since two people are being paid to do the work of one person. There are claims that quality is improved, but formal inspections combined with static analysis achieve much higher quality for much lower costs.

Another agile variation, extreme programming, in which test cases are created before the code itself is written has proven to be fairly successful for both quality and productivity, compared to traditional waterfall methods. However both TSP and RUP are just as good and even better for large systems. Another successful variation on agile is Disciplined agile development (DAD) which expands the agile concept up above 5,000 function points.

There are more than 80 available methodologies circa 2016 and many are good; some are better than agile for large systems; some older methods such as waterfall and cowboy development are at the bottom of the effectiveness list and should be avoided on modern applications.

For major applications in the 10,000 function point size range and above the team software process (TSP) and the Rational unified process (RUP) have the best track records for successful projects and among the fewest failures. Table 5 ranks 50 current software development methodologies. The rankings show their effectiveness for small projects below 1,000 function points and for large systems above 10,000 function points. Table 1 is based on data from around 600 companies and 25,000 project results:

Table 5: Methodology Rankings for Small and Large Software Projects

	Small Projects < 1000 function points	Large Systems > 10,000 function points
1	**Agile scrum**	**TSP/PSP**
2	**Crystal**	**Reuse-Oriented**
3	**DSDM**	**Pattern-based**
4	**Feature driven (FDD)**	**IntegraNova**
5	**Hybrid**	**Product Line engineering**
6	**IntegraNova**	**Model-driven**
7	**Lean**	**DevOps**
8	**Mashup**	**Service-Oriented**
9	**Microsoft solutions**	**Specifications by Example**
10	**Model-driven**	**Mashup**
11	**Object-Oriented**	**Object-oriented**
12	**Pattern-based**	**Information engineering (IE)**
13	**Product Line engineering**	**Feature driven (FDD)**
14	**PSP**	**Microsoft solutions**
15	**Reuse-oriented**	**Structured development**
16	**Service-Oriented modeling**	**Spiral development**
17	**Specifications by Example**	**T-VEC**
18	**Structured development**	**Kaizen**
19	**Test-driven development (TDD)**	**RUP**
20	**CASE**	**Crystal**
21	Clean room	DSDM
22	Continuous development	Hybrid
23	DevOps	CASE
24	EVO	Global 24 hour
25	Information engineering (IE)	Continuous development
26	Legacy redevelopment	Legacy redevelopment
27	Legacy renovation	Legacy renovation
28	Merise	Merise
29	Open-source	Iterative
30	Spiral development	Legacy data mining
31	T-VEC	Custom by client
32	Kaizen	CMMI 3
33	Pair programming	Agile scrum
34	Reengineering	Lean
35	Reverse engineering	EVO
36	XP	Open-source
37	Iterative	Reengineering

38	Legacy data mining	V-Model
39	Prototypes - evolutionary	Clean room
40	RAD	Reverse engineering
41	RUP	Prototypes - evolutionary
42	TSP/PSP	RAD
43	V-Model	Prince 2
44	Cowboy	Prototypes - disposable
45	Prince 2	Test-driven development (TDD)
46	Waterfall	Waterfall
47	Global 24 hour	Pair programming
48	CMMI 3	XP
49	Prototypes - disposable	Cowboy
50	Anti patterns	Anti patterns

The green color highlights the methods with the most successful project outcomes. In general the large-system methods are "quality strong" methodologies that support inspections and rigorous quality control. Some of these are a bit "heavy" for small projects although quality results are good. However the overhead of some rigorous methods tends to slow down small projects.

Starting in 2014 and expanding fairly rapidly is the new "software engineering methods and theory" or SEMAT approach. This is not a "methodology" per se but new way of analyzing software engineering projects and applications themselves.

SEMAT has little or no empirical data as this article is written but the approach seems to have merit. The probable impact, although this is not yet proven, will be a reduction in software defect potentials and perhaps an increase in certified reusable components.

Unfortunately SEMAT seems to be aimed at custom designs and manual development of software, both of which are intrinsically expensive and error-prone. SEMAT would be better used for increasing the supply of certified reusable components. As SEMAT usage expands it will be interesting to measure actual results, which to date are purely theoretical.

Quantifying Software Excellence

Because the software industry has a poor track record for measurement, it is useful to show what "excellence" means in quantified terms.

Excellence in software quality combines defect potentials of no more than 2.50 bugs per function point combined with defect removal efficiency (DRE) of 99.00%. This means that delivered defects will not exceed 0.025 defects per function point.

By contrast current average values circa 2016 are about 3.00 to 5.00 bugs per function point for defect potentials and only 90% to 94% DRE, leading to as many as 0.50 bugs

per function point at delivery. There are projects that top 99.00% percent but the distribution is less than 5% of U.S. projects top 99% in DRE as of 2016.

Poor projects which are likely to fail and end up in court for poor quality or breach of contract often have defect potentials of > 6.00 per function point combined with DRE levels < 85%. Some poor projects deliver > 0.75 bugs per function point and also excessive security flaws.

Excellence in software productivity and development schedules are not fixed values but varies with the size of the applications. Table 6 shows two "flavors" of productivity excellence: 1) the best that can be accomplished with 10% reuse and 2) the best that can be accomplished with 50% reuse:

Table 6: Excellent Productivity with Varying Quantities of Certified Reuse

	Schedule Months	Staffing	Effort Months	FP per Month
With < 10% certified reuse				
100 function points	4.79	1.25	5.98	16.71
1,000 function points	13.80	6.25	86.27	11.59
10,000 function points	33.11	57.14	1,892.18	5.28
100,000 function points	70.79	540.54	38,267.34	2.61
With 50% certified reuse				
100 function points	3.98	1.00	3.98	25.12
1,000 function points	8.51	5.88	50.07	19.97
10,000 function points	20.89	51.28	1,071.43	9.33
100,000 function points	44.67	487.80	21,789.44	4.59

As can be seen from table 6, software reuse is the most important technology for improving software productivity and quality by really significant amounts. Methods, tools, CMMI levels, SEMAT, and other minor factors are certainly beneficial. However so long as software applications are custom designed and hand coded software will remain an expensive craft and not a true professional occupation.

The Metaphor of Technical Debt

Ward Cunningham's interesting metaphor of "technical debt" has become a popular topic in the software industry. The concept of technical debt is that in order to get software released in a hurry, short cuts and omissions occur that will need to repaired after release, for much greater cost; i.e. like interest builds up on a loan.

Although the metaphor has merit, it is not yet standardized and therefore can vary widely. In fact a common question at conferences is "what do you include in technical debt?"

Technical debt is not a part of standard costs of quality. There are some other topics that are excluded also. The most important and also the least studied are "*consequential damages*" or actual financial harm to clients of buggy software. These show up in lawsuits against vendors and are known to attorneys and expert witnesses, but otherwise not widely published.

A major omission from technical debt circa 2016 is the cost of cyber-attacks and recovery from cyber-attacks. In cases where valuable data are stolen cyber-attack costs can be more expensive than total development costs for the attacked application.

Another omission from both cost of quality and technical are the costs of litigation and damage awards when software vendors or outsourcers are sued for poor quality. The final table in this report puts all of these costs together to show the full set of costs that might occur for excellent quality, average quality, and poor quality. Note that table 7 uses "defects per function point" for the quality results:

Table 7: Technical Debt and Software Quality for 1,000 function points

	High Quality	**Average Quality**	**Poor Quality**
Defect potential	**2**	4	**6**
Removal efficiency	**99.00%**	92.00%	**80.00%**
Delivered defects	**0.02**	0.32	**1.2**
Post-release defect repair $	**$5,000**	$60,000	**$185,000**
Technical debt problems	**1**	25	**75**
Technical debt costs	**$1,000**	$62,500	**$375,000**
Excluded from technical debt			

Consequential damages	**$0.00**	$281,250	**$2,437,500**
Cyber-attack costs	**$0.00**	$250,000	**$5,000,000**
Litigation costs	**$0.00**	$2,500,000	**$3,500,000**
Total Costs of Quality (COQ)	**$6,000**	**$3,153,750**	**$11,497,500**

As of early 2016 almost 85% of the true costs of poor quality software are invisible and not covered by either technical debt or standard "cost of quality" (COQ). No one has yet done a solid study of the damages of poor quality to clients and users but these costs are much greater than internal costs.

(This is a topic that should be addressed by both the CMMI and the SEMAT approach, although neither has studied consequential damages.)

No data has yet been published on the high costs of litigation for poor quality and project failures, or even the frequency of such litigation.

(The author has been an expert witness in 15 cases for project failure or poor quality, and therefore has better data than most on litigation frequencies and costs. Also the author's SRM tool has a standard feature that predicts probable litigation costs for both the plaintiff and defendant in breach of contract litigation.)

Table 7 illustrates two important but poorly understood facts about software quality economics:

1) ***High quality software is faster and cheaper to build than poor quality software; maintenance costs are many times cheaper; and technical debt is many times cheaper.***
2) ***Poor quality software is slower and more expensive to build than high quality software; maintenance costs are many times more expensive; and technical debt is many times more expensive.***

Companies that skimp on quality because they need to deliver software in a hurry don't realize that they are slowing down software schedules; not speeding them up.

High quality also causes little or no consequential damages to clients, and the odds of being sued are below 1%, as opposed to about 15% for poor quality software built by outsource vendors. Incidentally state governments seem to have more litigation for failing projects and poor quality than any other industry sector.

High quality projects are also less likely to experience cyber-attacks because many of these attacks are due to latent security flaws in deployed software. These flaws might have been eliminated prior to deployment if security inspections and security testing plus

static analysis had been used.

For software projects, high quality is more than free; it is one of the best investments companies can make. High quality has a large and positive return on investment (ROI). Poor quality software projects have huge risks of failure, delayed schedules, major cost overruns, and more than double the cost per function point compared to high quality.

Stages in Achieving Software Excellence

Readers are probably curious about the sequence of steps needed to move from "average" to "excellent" in software quality. They are also curious about the costs and schedules needed to achieve excellence. Following are short discussions of the sequence and costs needed for a company with about 1,000 software personnel to move from average to excellent results.

Stage 1: Quantify your current software results

In order to plan improvements rationally all companies should know their current status using effective quantified data points. This means that every company should measure and know these topics:

1. Defect potentials
2. Defect severity levels
3. Defects per function point
4. Defect detection efficiency (DDE)
5. Defect removal efficiency (DRE)
6. Cyclomatic complexity of all applications
7. Error-prone modules (EPM) in deployed software
8. Test coverage of all applications
9. Test cases and test scripts per function point
10. Duplicate or incorrect test cases in test libraries
11. Bad-fix injection rates (bugs in defect repairs)
12. The existence or absence of error-prone modules in operational software
13. Customer satisfaction with existing software
14. Defect repair turnaround
15. Technical debt for deployed software
16. Cost of quality (COQ)
17. Security flaws found before release and then after deployment
18. Current set of defect prevention, pre-test, and test quality methods in use
19. The set of software development methodologies in use for all projects
20. Amount of reusable materials utilized for software projects

For a company with 1,000 software personnel and a portfolio of perhaps 3,000 software applications this first stage can take from two to three calendar months. The effort would probably be in the range of 15 to 25 internal staff months, plus the use of external quality consultants during the fact-finding stage.

The most likely results will be the discovery that defect potentials top 3.5 per function point and defect removal efficiency (DRE) is below 92%. Other likely findings will include < 80% test coverage and cyclomatic complexity that might > 50 for key modules. Probably a dozen or more error-prone modules will be discovered. Quantitative goals for every software company should be to have defect potentials < 2.5 per function point combined with DRE levels > 97% for every software project, and above 99% for mission-critical software projects. Software reuse will probably be < 15% and mainly be code modules that are picked up informally from other applications.

The analogy for this stage would be like going to a medical clinic for a thorough annual medical check-up. The check-up does not cure any medical problems by itself, but it identifies the problems that physicians will need to cure, if any exist.

Once the current quality results have been measured and quantified, it is then possible to plan rational improvement strategies that will reduce defect potentials and raise defect removal efficiency to approximate 99% levels.

Stage 2: Begin to Adopt State of the Art Quality Tools and Methods

Software excellence requires more than just adopting a new method such as agile and assuming everything will get better. Software excellence is the result of a web of related methods and tools that are synergistic.

The second stage, which occurs as the first stage is ending, and perhaps overlaps the last month, is to acquire and start to use proven methods for defect prevention, pre-test defect removal, and formal testing.

This stage can vary by the nature and size of the software produced. Real-time and embedded applications will use different tools and methods compared to web and information technology applications. Large systems will use different methods than small applications. However a nucleus of common techniques is used for all software. These include the following:

Formal Sizing, Estimating, and Tracking

1) Use parametric estimation tools on projects > 250 function points
2) Carry out formal risk analysis before starting
3) Use formal tracking of progress, quality, and costs

Defect prevention

1. Joint application design (JAD)
2. Quality function deployment (QFD)
3. Requirements models
4. Formal reuse programs
5. Formal defect measurements
6. Data mining of legacy applications for lost requirements

7. Training and certification of quality personnel
8. Acquisition of defect measurements tools and methods
9. Formal methodology analysis and selection for key projects
10. Formal quality and defect estimation before projects start

Pre-test defect removal

1. Static analysis of all legacy applications
2. Static analysis of all new applications
3. Static analysis of all changes to applications
4. Inspections of key deliverables for key projects (requirements, design, code, etc.)
5. Automated proofs of correctness for critical features

Test defect removal

1. Formal test case design, often using design of experiments or cause-effect graphs
2. Acquisition of test coverage tools
3. Acquisition of cyclomatic complexity tools
4. Review of test libraries for duplicate or defective test cases
5. Formal training of test personnel
6. Certification of test personnel
7. Planning optimal test sequences for every key project
8. Measuring test coverage for all projects
9. Measuring cylomatic complexity for all code
10. Formal test and quality measures of all projects

This second stage normally lasts about a year and includes formal training of managers, development personnel, quality assurance personnel, test personnel, and other software occupation groups.

Because there is a natural tendency to resist changes, the best way of moving forward is to treat the new tools and methods as experiments. In other words, instead of directing that certain methods such as inspections be used, treat them as experiments and make it clear that if the inspections don't seem useful after trying them out, the teams will not be forced to continue with them. This is how IBM introduced inspections in the 1970's, and the results were so useful that inspections became a standard method without any management directives.

This second stage will take about a year for a company with 1,000 software personnel, and more or less time for larger or smaller organizations. Probably all technical personnel will receive at least a week of training, and so will project managers.

Probably the costs during this phase due to training and learning curves can top $1,000 per staff member. Some costs will be training; others will be acquisitions of tools. It is difficult to establish a precise cost for tools due to the availability of a large number of open-source tools that have no costs.

Improvements in quality will start to occur immediately during stage 2. However due to

learning curves, productivity will drop down slightly for the first 4 months due to having formal training for key personnel. But by the end of a year, productivity may be 15% higher than when the year started. Defect potentials will probably drop by 20% and defect removal efficiency (DRE) should go up by > 7% from the starting point, and top 95% for every project.

Stage 3: Continuous Improvements Forever

Because stages 1 and 2 introduce major improvements, some interesting sociological phenomena tend to occur. One thing that may occur is that the technical and management leaders of stages 1 and 2 are very likely to get job offers from competitive companies or from other divisions in large corporations.

It sometimes happens that if the stage 1 and 2 leaders are promoted or change jobs, their replacements may not recognize the value of the new tools and methods. For example many companies that use inspections and static analysis find that defects are much reduced compared to previous years.

When quality improves significantly unwise managers may say, "*why keep using inspections and static analysis when they are not finding many bugs?*" Of course if the inspections and static analysis stop, the bug counts will soon start to climb back up to previous levels and DRE will drop down to previous levels.

In order to keep moving ahead and staying at the top, formal training and formal measurements are both needed. Annual training is needed, and also formal training of new personnel and new managers. Companies that provide 5 or more days of training for software personnel have higher annual productivity than companies with zero days of training.

When the ITT Corporation began a successful 4-year improvement program, one of the things that was part of their success was an annual report for corporate executives. This report was produced on the same schedule as the annual corporate financial report to shareholders; i.e. in the first quarter of the next fiscal year.

The ITT annual reports showed accomplishments for the prior year; comparisons to earlier years; and projected accomplishments for the following year. Some of the contents of the annual reports included:

1. Software personnel by division
2. Software personnel by occupation groups
3. Year-by-year cost of quality (COQ)
4. Total costs of software ownership (TCO)
5. Changes in software personnel by year for three years
6. Average and ranges of defect potentials
7. Average and ranges of defect removal efficiency (DRE)
8. Three-year running averages of defect potentials and DRE

9. Customer satisfaction year by year
10. Plans for the next fiscal year for staffing, costs, quality, etc.

ITT was a large corporation with over 10,000 software personnel located in a number of countries and more than 25 software development labs. As a result the overall corporate software report was a fairly large document of about 50 pages in size.

For a smaller company with a staffing of about 1,000 personnel, the annual report would probably be in the 20-page size range.

Once software is up to speed and combines high quality and high productivity, that opens up interesting business questions about the best use of the savings. For example ITT software personnel had been growing at more than 5% per year for many years. Once quality and productivity improved, it was clear that personnel growth was no longer needed. In fact the quality and productivity were so good after a few years that perhaps 9,000 instead of 10,000 could build and maintain all needed software.

Some of the topics that need to be considered when quality and productivity improve are related to what is the best use of resources no longer devoted to fixing bugs. Some of the possible uses include:

- Reduce corporate backlogs to zero by tackling more projects per year.

- Move into new kinds of applications using newly available personnel no longer locked into bug repairs.

- Allow natural attrition to lower overall staffing down to match future needs.

For commercial software companies expanding into new kinds of software and tackling more projects per year are the best use of available personnel that will be freed up when quality improves.

For government software or for companies that are not expanding their businesses, then probably allowing natural attrition to reduce staffing might be considered. For large organizations, transfers to other business units might occur.

One thing that would a sociological disaster would be to have layoffs due to the use of improved technologies that reduced staffing needs. In this case resistance to changes and improvements would become a stone wall and progress would stop cold.

Since most companies have large backlogs of applications that are awaiting development, and since most leading companies have needs to expand software into new areas, the best overall result would be to use the available personnel for expansion

Stage three will run for many years. The overall costs per function point should be about 30% lower than before the improvement program started. Overall schedules should be

about 25% shorter than before the improvement program started.

Defect potentials will be about 35% lower than when the improvement program started and corporate defect removal efficiency should top 97% for all projects and 99% for mission critical projects.

Going Beyond Stage 3 into Formal Reuse Programs

As mentioned previously in this report, custom designs and manual coding are intrinsically expensive and error-prone no matter what methodologies are used and what programming languages are used.

For companies that need peak performance, moving into a full and formal software reuse program can achieve results even better than Stage 3.

Summary and Conclusions

Because software is the driving force of both industry and government operations, it needs to be improved in terms of both quality and productivity. The most powerful technology for making really large improvements in both quality and productivity will be from eliminating costly custom designs and labor-intensive hand coding, and moving towards manufacturing software applications from libraries of well-formed standard reusable components that approach zero-defect quality levels.

Today's best combinations of methods, tools, and programming languages are certainly superior to waterfall or cowboy development using unstructured methods and low-level languages. But even the best current methods still involve error-prone custom designs and labor-intensive manual coding.

REFERENCES AND READINGS

Abran, A. and Robillard, P.N.; "Function Point Analysis, An Empirical Study of its Measurement Processes"; IEEE Transactions on Software Engineering, Vol 22, No. 12; Dec. 1996; pp. 895-909.

Austin, Robert d:; Measuring and Managing Performance in Organizations; Dorset House Press, New York, NY; 1996; ISBN 0-932633-36-6; 216 pages.

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

Bogan, Christopher E. and English, Michael J.; Benchmarking for Best Practices; McGraw Hill, New York, NY; ISBN 0-07-006375-3; 1994; 312 pages.

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

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

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

Curtis, Bill, Hefley, William E., and Miller, Sally; People Capability Maturity Model; Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA; 1995.

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

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

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

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

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

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

pages.

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

Grady, Robert B.; Successful Process Improvement; Prentice Hall PTR, Upper Saddle River, NJ; ISBN 0-13-626623-1; 1997; 314 pages.

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

IFPUG Counting Practices Manual, Release 4, International Function Point Users Group, Westerville, OH; April 1995; 83 pages.

Jacobsen, Ivar, Griss, Martin, and Jonsson, Patrick; Software Reuse - Architecture, Process, and Organization for Business Success; Addison Wesley Longman, Reading, MA; ISBN 0-201-92476-5; 1997; 500 pages.

Jacobsen, Ivar et al; The Essence of Software Engineering; Applying the SEMAT Kernel; Addison Wesley Professional, 2013.

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

Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality, Addison Wesley Longman, Reading, MA; 2011.

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

Jones, Capers; Software Engineering Best Practices; McGraw Hill, New York, NY; 2010.

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

Jones, Capers; "A Ten-Year Retrospective of the ITT Programming Technology Center"; Software Productivity Research, Burlington, MA; 1988.

Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008.

Jones, Capers; Software Engineering Best Practices; McGraw Hill, 1st edition 2010.

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 Assessments, Benchmarks, and Best Practices; Addison Wesley Longman, Boston, MA; 2000 (due in May of 2000); 600 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; The Economics of Object-Oriented Software; Software Productivity Research, Burlington, MA; April 1997; 22 pages.

Jones, Capers; Becoming Best in Class; Software Productivity Research, Burlington, MA; January 1998; 40 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.

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

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

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

McMahon, Paul; 15 Fundamentals for Higher Performance in Software Development; PEM Systems 2014.

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

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

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

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

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

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

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

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

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

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

Rubin, Howard (Editor); The Software Personnel Shortage; Rubin Systems, Inc.; Pound Ridge, NY; 1998.

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

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

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

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

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

Umbaugh, Robert E. (Editor); Handbook of IS Management; (Fourth Edition); Auerbach

Publications, Boston, MA; ISBN 0-7913-2159-2; 1995; 703 pages.

Weinberg, Dr. Gerald; Quality Software Management - Volume 2 First-Order Measurement; Dorset House Press, New York, NY; ISBN 0-932633-24-2; 1993; 360 pages.

Wiegers, Karl A; Creating a Software Engineering Culture; Dorset House Press, New York, NY; 1996; ISBN 0-932633-33-1; 358 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.

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

Zvegintzov, Nicholas; Software Management Technology Reference Guide; Dorset House Press, New York, NY; 1994; ISBN 1-884521-01-0; 240 pages.