SCORING AND EVALUATING
SOFTWARE METHODS, PRACTICES, AND RESULTS

Version 5.0 June 14, 2012

Abstract

Software engineering and software project management are complex activities. Software development, maintenance, and software management have dozens of methodologies and hundreds of tools available that are beneficial. In addition, there are quite a few methods and practices that have been shown to be harmful, based on depositions and court documents in litigation for software project failures.

In order to evaluate the effectiveness or harm of these numerous and disparate factors, a simple scoring method has been developed. The scoring method runs from +10 for maximum benefits to -10 for maximum harm.

The scoring method is based on quality and productivity improvements or losses compared to a mid-point. The mid point is traditional waterfall development carried out by projects at about level 1 on the Software Engineering Institute capability maturity model (CMMI) using low-level programming languages. Methods and practices that improve on this mid point are assigned positive scores, while methods and practices that show declines are assigned negative scores.

The data for the scoring comes from observations among about 150 Fortune 500 companies, some 50 smaller companies, and 30 government organizations. Negative scores also include data from 15 lawsuits.

The scoring method does not have high precision and the placement is somewhat subjective. However, the scoring method does have the advantage of showing the range of impact of a great many variable factors. This article is based on the author's book Software Engineering Best Practices published by McGraw Hill at the end of 2009. Some new data is taken from The Economics of Software Quality published by Addison Wesley in 2011.

Capers Jones
Vice President and CTO, Namcook Analytics LLC
Capers.Jones3@gmail.com

INTRODUCTION:
EVALUATING SOFTWARE METHODS, PRACTICES, AND RESULTS

Software development and software project management have dozens of methods, hundreds of tools, and scores of practices. Many of these are beneficial, but many are harmful too. There is a need to be able to evaluate and rank many different topics using a consistent scale.

To deal with this situation a scoring method has been developed that allows disparate topics to be ranked using a common scale. Methods, practices, and results are scored using a scale that runs from +10 to -10 using the criteria shown in table 1.

Both the approximate impact on productivity and the approximate impact on quality are included. The scoring method can be applied to specific ranges such as 1000 function points or 10,000 function points. It can also be applied to specific types of software such as Information Technology, Web application, commercial software, military software, and several others.

Table 1: Scoring Ranges for Software Methodologies and Practices

Score	Productivity Improvement	Quality Improvement
10	25%	35%
9	20%	30%
8	17%	25%
7	15%	20%
6	12%	17%
5	10%	15%
4	7%	10%
3	3%	5%
2	1%	2%
1	0%	0%
0	0%	0%
-1	0%	0%
-2	-1%	-2%
-3	-3%	-5%
-4	-7%	-10%
-5	-10%	-15%
-6	-12%	-17%
-7	-15%	-20%
-8	-17%	-25%
-9	-20%	-30%
-10	-25%	-35%

The midpoint or "average" against which improvements are measured are traditional application development methods such as "waterfall" development performed by organizations that either don't use the Software Engineering Institute's capability maturity model or are at level 1. Low-level programming languages are also assumed. This fairly primitive combination remains more or less the most widely used development method even in 2012.

One important topic needs to be understood. Quality needs to be improved faster and to a higher level than productivity in order for productivity to improve at all. The reason for this is that finding and fixing bugs is overall the most expensive activity in software development. Quality leads and productivity follows. Attempts to improve productivity without improving quality first are not effective.

For software engineering a serious historical problem has been that measurement practices are so poor that quantified results are scarce. There are many claims for tools, languages, and methodologies that assert each should be viewed as a "best practice." But empirical data on their actual effectiveness in terms of quality or productivity has been scarce. Three points need to be considered.

The first point is that software applications vary by many orders of magnitude in size. Methods that might be ranked as "best practices" for small programs of 1,000 function points in size may not be equally effective for large systems of 100,000 function points in size.

The second point is that software engineering is not a "one size fits all" kind of occupation. There are many different forms of software such as embedded applications, commercial software packages, information technology projects, games, military applications, outsourced applications, open-source applications and several others. These various kinds of software applications do not necessarily use the same languages, tools, or development methods.

The third point is that tools, languages, and methods are not equally effective or important for all activities. For example a powerful programming language such as Objective C will obviously have beneficial effects on coding speed and code quality. But which programming language is used has no effect on requirements creep, user documentation, or project management. Therefore the phrase "best practice" also has to identify which specific activities are improved. This is complicated because activities include development, deployment, and post-deployment maintenance and enhancements. Indeed, for large applications development can take up to five years, installation can take up to one year, and usage can last as long as 25 years before the application is finally retired. Over the course of more than 30 years there will be hundreds of activities.

The result of these various factors is that selecting a set of "best practices for software engineering" is a fairly complicated undertaking. Each method, tool, or language needs to be evaluated in terms of its effectiveness by size, by application type, and by activity.

Overall Rankings of Methods, Practices, and Sociological Factors

In order to be considered a "best practice" a method or tool has to have some quantitative proof that it actually provides value in terms of quality improvement, productivity improvement, maintainability improvement, or some other tangible factors.

Looking at the situation from the other end, there are also methods, practices, and social issues have demonstrated that they are harmful and should always be avoided. For the most part the data on harmful factors comes from depositions and court documents in litigation.

In between the "good" and "bad" ends of this spectrum are practices that might be termed "neutral." They are sometimes marginally helpful and sometimes not. But in neither case do they seem to have much impact.

Although the author's book Software Engineering Best Practices dealt with methods and practices by size and by type, it might be of interest to show the complete range of factors ranked in descending order, with the ones having the widest and most convincing proof of usefulness at the top of the list. Table 2 lists a total of 220 methodologies, practices, and social issues that have an impact on software applications and projects.

The average scores shown in table 2 are actually based on the composite average of six separate evaluations:

1. Small applications < 1000 function points
2. Medium applications between 1000 and 10,000 function points
3. Large applications > 10,000 function points
4. Information technology and web applications
5. Commercial, systems, and embedded applications
6. Government and military applications

The data for the scoring comes from observations among about 150 Fortune 500 companies, some 50 smaller companies, and 30 government organizations and around 13,000 total projects. Negative scores also include data from 15 lawsuits.

The scoring method does not have high precision and the placement is somewhat subjective. However, the scoring method does have the advantage of showing the range of impact of a great many variable factors. This article is based on the author's two recent books: Software Engineering Best Practices published by McGraw Hill in 2009 and The Economics of Software Quality published by Addison Wesley in 2011..

The full spreadsheet with all six rankings is quite large and complex, so only the overall average results are shown here:

Table 2: Evaluation of Software Methods, Practices, Tools

	Best Practices	
1	Reusability (> 85% zero-defect materials)	9.65
2	Requirements patterns - InteGreat	9.50
3	Defect potentials < 3.00 per function point	9.35
4	Requirements modeling (T-VEC)	9.33
5	Defect removal efficiency > 95%	9.32
6	Personal Software Process (PSP)	9.25
7	Team Software Process (TSP)	9.18
8	Automated static analysis - code	9.17
9	Mathematical test case design (Hexawise)	9.17
10	Inspections (code)	9.15
11	Measurement of defect removal efficiency	9.08
12	Hybrid (CMM+TSP/PSP+others)	9.06
13	Automated static analysis - text	9.00
14	FOG readability index - requirements	9.00
15	Reusable feature certification	9.00
16	Reusable feature change controls	9.00
17	Reusable feature recall method	9.00
18	Reusable feature warranties	9.00
19	Reusable source code (zero defect)	9.00
	Very Good Practices	
20	Continuous integration	8.83
21	Early estimates of defect potentials	8.83
22	FLESCH readability index - requirements	8.83
23	Object-oriented development (OO)	8.83
24	Automated security testing	8.58
25	Automated UML static analysis	8.50
26	Measurement of bad-fix injections	8.50
27	Reusable test cases (zero defects)	8.50
28	Test case inspections	8.50
29	Prince2	8.50
30	Formal security analysis	8.43
31	ITIL - customized	8.42
32	Agile development with SCRUM	8.41
33	Inspections (requirements)	8.40
34	Time boxing	8.38
35	Activity-based productivity measures	8.33
36	Reusable designs (scalable)	8.33
37	Formal risk management	8.27
38	Automated defect tracking tools	8.17
39	Measurement of defect origins	8.17
40	Benchmarks against industry data	8.15
41	Function point analysis (high-speed)	8.15
42	Formal progress reports (weekly)	8.06
43	Formal measurement programs	8.00

44	Project offices - automated	8.00
45	Reusable architecture (scalable)	8.00
46	Certification - vendor (Apple, Microsoft, etc.)	8.00
47	Inspections (design)	7.94
48	Lean Six-Sigma	7.94
49	Six-sigma for software	7.94
50	Automated cost estimating tools	7.92
51	Automated maintenance work benches	7.90
52	Formal cost tracking reports	7.89
53	ITIL - normal	7.83
54	Requirements modeling (IntegraNova)	7.83
55	Formal test plans	7.81
56	Automated unit testing	7.75
57	Automated sizing tools (function points)	7.73
58	Scrum session (daily)	7.70
59	Automated configuration control	7.69
60	Reusable requirements (scalable)	7.67
61	Automated project management tools	7.63
62	Formal requirements analysis	7.63
63	Data mining for business rule extraction	7.60
64	Function point analysis (pattern matches)	7.58
65	High-level languages (current)	7.53
66	Automated quality and risk prediction	7.53
67	Automated ERP estimates	7.50
68	Reusable tutorial materials	7.50
69	Function point analysis (IFPUG)	7.37
70	Measurement of requirements changes	7.37
71	Formal architecture for large applications	7.36
72	Best-practice analysis before start	7.33
73	Reusable feature catalog	7.33
74	Quality function deployment (QFD)	7.32
75	Specialists for key skills	7.29
76	Joint Application Design (JAD)	7.27
77	Automated test coverage analysis	7.23
78	Re-estimating for requirements changes	7.17
79	Measurement of defect severity levels	7.13
80	Formal SQA team	7.10
81	Inspections (test materials)	7.04
82	Automated requirements analysis	7.00
83	DMAIC	7.00
84	Reusable construction plans	7.00
85	Reusable HELP information	7.00
86	Reusable test scripts	7.00

	Good Practices	
87	Rational Unified Process (RUP)	6.98
88	Automated deployment support	6.87
89	Automated cyclomatic complexity analysis	6.83
90	Forensic analysis of cancelled projects	6.83
91	Reusable reference manuals	6.83
92	Automated documentation tools	6.79
93	Capability Maturity Model (CMMI Level 5)	6.79
94	Annual training (technical staff)	6.67
95	Metrics conversion (automated)	6.67
96	Change review boards	6.62
97	Formal Governance	6.58
98	Automated test library control	6.50
99	Formal scope management	6.50
100	Project offices - manual	6.50
101	Annual training (managers)	6.33
102	Dashboard-style status reports	6.33
103	Extreme programming (XP)	6.28
104	Service-Oriented Architecture (SOA)	6.26
105	Automated requirements tracing	6.25
106	Total Cost of Ownership (TCO) measures	6.18
107	Automated performance analysis	6.17
108	Baselines for process improvement	6.17
109	Use cases	6.17
110	Automated test case generation	6.00
111	User satisfaction surveys	6.00
112	Formal project office	5.88
113	Automated modeling/simulation	5.83
114	Certification (six sigma)	5.83
115	Outsourcing (maintenance => CMMI 3)	5.83
116	Capability Maturity Model (CMMI Level 4)	5.79
117	Certification (software quality assurance)	5.67
118	Outsourcing (development => CMM 3)	5.67
119	Value analysis (intangible value)	5.67
120	Root-cause analysis	5.50
121	Total Cost of Learning (TOL) measures	5.50
122	Cost of quality (COQ)	5.42
123	Embedded users in team	5.33
124	Normal structured design	5.17
125	Capability Maturity Model (CMMI Level 3)	5.06
126	Earned-value measures	5.00
127	Unified Modeling Language (UML)	5.00
128	Value analysis (tangible value)	5.00

	Fair Practices	
129	Normal maintenance activities	4.54
130	Rapid application development (RAD)	4.54
131	Certification (function points)	4.50
132	Function point analysis (Finnish)	4.50
133	Function point analysis (Netherlands)	4.50
134	Partial code reviews	4.42
135	Automated restructuring	4.33
136	Function point analysis (COSMIC)	4.33
137	Function point analysis (unadjusted)	4.33
138	Partial design reviews	4.33
139	Team Wiki communications	4.33
140	Function points (micro .001 to 10)	4.17
141	Automated daily progress reports	4.08
142	User stories	3.83
143	Outsourcing (offshore => CMM 3)	3.67
144	Goal-question metrics	3.50
145	Certification (project managers)	3.33
146	Refactoring	3.33
147	Manual document production	3.17
148	Capability Maturity Model (CMMI Level 2)	3.00
149	Certification (test personnel)	2.83
150	Pair programming	2.83
151	Clean-room development	2.50
152	Formal design languages	2.50
153	ISO Quality standards	2.00
	Neutral Practices	
154	Function point analysis (backfiring)	1.83
155	Use Case points	1.67
156	Normal customer support	1.50
157	Partial governance (low risk projects)	1.00
158	Object-oriented metrics	0.33
159	Manual testing	0.17
160	Outsourcing (development < CMM 3)	0.17
161	Story points	0.17
162	Low-level languages (current)	0.00
163	Outsourcing (maintenance < CMM 3)	0.00
	Unsafe Practices	
164	Waterfall development	(0.33)
165	Manual change control	(0.50)
166	Manual test library control	(0.50)
167	Reusability (average quality materials)	(0.67)
168	Capability Maturity Model (CMMI Level 1)	(1.50)
169	Informal progress tracking	(1.50)
170	Outsourcing (offshore < CMM 3)	(1.67)

171	Inadequate test library control	(2.00)
172	End-user development	(2.00)
173	Generalists instead of specialists	(2.50)
174	Manual cost estimating methods	(2.50)
175	Inadequate measurement of productivity	(2.67)
176	Cost per defect metrics	(2.83)
177	Inadequate customer support	(2.83)
178	Friction between stakeholders and team	(3.50)
179	Informal requirements gathering	(3.67)
	Hazardous Practices	
180	Lines of code metrics (logical LOC)	(4.00)
181	Inadequate governance	(4.17)
182	Lines of code metrics (physical LOC)	(4.50)
183	Partial productivity measures (coding)	(4.50)
184	Inadequate sizing	(4.67)
185	High-level languages (obsolete)	(5.00)
186	Inadequate communications among team	(5.33)
187	Inadequate change control	(5.42)
188	Inadequate value analysis	(5.50)
189	Cowboy programming	(5.67)
190	Friction/antagonism among team members	(6.00)
191	Inadequate cost estimating methods	(6.04)
192	Inadequate risk analysis	(6.17)
193	Low-level languages (obsolete)	(6.25)
194	Government mandates (short lead times)	(6.33)
195	Inadequate testing	(6.38)
196	Friction/antagonism among management	(6.50)
197	Inadequate communications with stakeholders	(6.50)
198	Inadequate measurement of quality	(6.50)
199	Inadequate problem reports	(6.67)
200	Error-prone modules in applications	(6.83)
201	Friction/antagonism among stakeholders	(6.83)
202	Failure to estimate requirements changes	(6.85)
	Worst Practices	
203	Inadequate defect tracking methods	(7.17)
204	Layoffs/loss of key personnel	(7.33)
205	Rejection of estimates for business reasons	(7.33)
206	Inadequate inspections	(7.42)
207	Inadequate security controls	(7.48)
208	Excessive schedule pressure	(7.50)
209	Inadequate progress tracking	(7.50)
210	Litigation (non-compete violation)	(7.50)
211	Defective test cases	(7.67)
212	Inadequate cost tracking	(7.75)
213	Litigation (breach of contract)	(8.00)
214	Defect potentials > 6.00 per function point	(9.00)
215	Reusability (high defect volumes)	(9.17)

216	Defect removal efficiency < 85%	(9.18)
217	Litigation (poor quality/damages)	(9.50)
218	Litigation (security flaw damages)	(9.50)
219	Litigation (intellectual property theft)	(10.00)
220	Litigation (patent violation)	(10.00)

It should be realized that table 2 is a work in progress and changes frequently. Also, the value of table 2 is not in the precision of the rankings, which are somewhat subjective, but in the ability of the simple scoring method to show the overall sweep of many disparate topics using a single scale.

Table 2 is often used as a quick evaluation method for software organizations and software projects. From interviews with project teams and software managers, the methods actually deployed are checked off on table 2. Then the numeric scores from table 2 are summed and averaged.

A leading company will deploy methods that, when summed, total to more than 250 and average more than 5.5. Lagging organizations and lagging projects will sum to less than 100 and average below 4.0. The worst average encountered so far is was only 1.8 and that was done as background to a lawsuit for breach of contract. The vendor who was also the defendant was severely behind in the use of effective methods and practices.

Note that the set of factors included are a mixture. They include full development methods such as Team Software Process (TSP), partial methods such as Quality Function Deployment (QFD). They include specific practices such as "inspections" of various kinds, and also social issues such as friction between stakeholders and developers. They also include metrics such as "lines of code" which is ranked as a harmful factor because this metric penalizes high-level languages and distorts both quality and productivity data. What all these things they have in common is that they either improve or degrade quality and productivity.

Since programming languages are also significant, it might be asked why specific languages such as Java, Ruby, or Objective C are not included. This is because as of 2012 more than 2,500 programming languages exist, and new languages are being created at a rate of about one every calendar month.

In addition, a majority of large software applications utilize several languages at the same time, such as JAVA and HTML, or combinations that may top a dozen languages in the same applications. There are too many languages and they change far too rapidly for an evaluation to be useful for more than a few months of time. Therefore languages are covered only in a general way: are they high-level or low-level, and are they current languages or "dead" languages no longer in use for new development.

Unfortunately a single list of values averaged over three different size ranges and multiple types of applications does not illustrate the complexity of best-practice analysis. Shown below are examples of 30 best practices for small applications of 1000 function points and for large systems of 10,000 function points. As can be seen, the two lists have very different patterns of best practices.

The flexibility of the Agile methods are a good match for small applications, while the rigor of Team Software Process (TSP) and Personal Software Process (PSP) are a good match for the difficulties of large-system development.

Table 3: Best Practice Differences between 1000 and 10,000 Function Points

	Small (1000 function points)	Large (10,000 function points)
1	Agile development	Reusability (> 85% zero-defect materials)
2	High-level languages (current)	Defect potentials < 3.00 per function point
3	Extreme programming (XP)	Formal cost tracking reports
4	Personal Software Process (PSP)	Inspections (requirements)
5	Reusability (> 85% zero-defect materials)	Formal security analysis
6	Automated static analysis	Measurement of defect removal efficiency
7	Time boxing	Team Software Process (TSP)
8	Reusable source code (zero defect)	Function point analysis (high-speed)
9	Reusable feature warranties	Capability Maturity Model (CMMI Level 5)
10	Reusable feature certification	Automated security testing
11	Defect potentials < 3.00 per function point	Inspections (design)
12	Reusable feature change controls	Defect removal efficiency > 95%
13	Reusable feature recall method	Inspections (code)
14	Object-oriented development (OO)	Automated sizing tools (function points)
15	Inspections (code)	Hybrid (CMM+TSP/PSP+others)
16	Defect removal efficiency > 95%	Automated static analysis
17	Hybrid (CMM+TSP/PSP+others)	Personal Software Process (PSP)
18	Scrum session (daily)	Automated cost estimating tools
19	Measurement of defect removal efficiency	Measurement of requirements changes
20	Function point analysis (IFPUG)	Service-Oriented Architecture (SOA)
21	Automated maintenance work benches	Automated quality and risk prediction
22	Early estimates of defect potentials	Benchmarks against industry data
23	Team Software Process (TSP)	Quality function deployment (QFD)
24	Embedded users in team	Formal architecture for large applications
25	Benchmarks against industry data	Automated defect tracking tools
26	Measurement of defect severity levels	Reusable architecture (scalable)
27	Use cases	Formal risk management
28	Reusable test cases (zero defects)	Activity-based productivity measures
29	Automated security testing	Formal progress reports (weekly)
30	Measurement of bad-fix injections	Function point analysis (pattern matches)

It is useful discuss polar opposites and both best practices and also show worst practices too. The definition of a "worst practice" is a method or approach that has been proven to cause harm to a significant number of projects that used it. The word "harm" means either degradation of quality, reduction of productivity, or concealing the true status of projects. In addition "harm" also includes data that is so inaccurate that it leads to false conclusions about economic value.

Each of the harmful methods and approaches individually has been proven to cause harm in a significant number of applications that used them. This is not to say that they always fail. Sometimes rarely they may even be useful. But in a majority of situations they do more harm than good in repeated trials.

What is a distressing aspect of the software industry is that bad practices seldom occur in isolation. From looking the depositions and court documents of lawsuits for projects that were cancelled or never operated effectively, it usually happens that multiple worst practices are used concurrently.

From data and observations on the usage patterns of software methods and practices, it is distressing to note that practices in the harmful or worst set are actually found on about 65% of U.S. Software projects as noted when doing assessments. Conversely, best practices that score 9 or higher have only been noted on about 14% of U.S. Software projects. It is no wonder that failures far outnumber successes for large software applications!

From working as an expert witness in a number of breach-of-contract lawsuits, many harmful practices tend to occur repeatedly. These collectively are viewed by the author as candidates for being deemed "professional malpractice." The definition of professional malpractice is something that causes harm which a trained practitioner should know is harmful and therefore should avoid using it.

Following are 30 issues that have caused trouble so often that the author views them as professional malpractice, primarily if they occur for applications in the 10,000 function point size range. That is the range where failures outnumber successes and where litigation is distressingly common. Only one of 15 lawsuits where the author worked as an expert witness was smaller than 10,000 function points.

Table 4 Candidates for Classification as "Professional Malpractice"

1	Defect removal efficiency < 85%
2	Defect potentials > 6.00 per function point
3	Reusability (uncertified with high defect volumes)
4	Inadequate cost tracking with "leakage" > 40% of actual costs
5	Excessive schedule pressure by clients and executives
6	Inadequate or deceptive progress tracking that conceals problems
7	Inadequate security controls
8	Inadequate inspections of requirements, design, and code
9	Inadequate defect tracking methods that starts late
10	Failure to estimate requirements changes
11	Error-prone modules in applications
12	Inadequate problem reports
13	Inadequate measurement of quality
14	Rejection of estimates for business reasons by clients or executives
15	Inadequate testing with low coverage
16	Inadequate risk analysis prior to funding
17	Inadequate cost estimating methods
18	Inadequate value analysis
19	Inadequate change control
20	Inadequate sizing prior to funding
21	Partial productivity measures that concentrates on coding

22 Lines of code metrics (LOC) for economic analysis
23 Inadequate governance by corporate executives
24 Inadequate requirements gathering
25 Cost per defect metrics
26 Inadequate customer support
27 Inadequate measurement of productivity
28 Generalists instead of specialists for large systems
29 Manual cost estimating methods for large systems
30 Inadequate test library control

It is unfortunate that several of these harmful practices, such as "cost per defect" and "lines of code" are still used for hundreds of projects without the users even knowing that "cost per defect" penalizes quality and "lines of code" penalizes high-level languages.

Collectively many or most of these 30 harmful practices are noted in more than 75% of software applications =>10,000 function points in size. Below 1,000 function points the significance of many of these decline and they would drop out of the malpractice range.

SUMMARY AND CONCLUSIONS

The phrase "software engineering" is actually a misnomer. Software development is not a recognized engineering field. Worse, large software applications fail and run late more often than they succeed.

There are countless claims of tools and methods that are advertised as improving software, but a severe shortage of empirical data on things that really work. There is also a shortage of empirical data on things that cause harm.

The simple scoring method used in this article attempts to provide at least a rough correlation between methods and practices and their effectiveness, quality, and productivity. The current results are somewhat subjective and may change as new data becomes available. However, the scoring method does illustrate a wide range of results from extremely valuable to extremely harmful.

REFERENCES AND SUGGESTED READINGS

Bundschuh, Manfred and Dekkers, Carol; The IT Measurement Compendium; Springer-Verlag, Heidelberg, Deutschland; ISBN 978-3-540-68187-8; 2008.

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 and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley, Boston, MA; 2012; ISBN 10: 0-13-258220-1; 587 pages.

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

Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008; ISBN 978-0-07-150244-3; 575 pages; 3rd edition due in the Spring of 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.

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

McConnell, Code Complete; Microsoft Press, Redmond, WA; 1993; ISBN 13-978-1556154843; 886 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

Process Fusion: www.process-fusion.net

Project Management Institute (www.PMI.org)

Software Engineering Institute (SEI): www.SEI.org

Software Productivity Research (SPR): www.SPR.com