Return on Investment
Using Software Inspections

Don O'Neill
Independent Consultant

Outline

Figures

6. Return on Investment

Return on Investment
Using Software Inspections

Keywords
Additional cost multiplier
Cost to repair
Defect detection cost
Defect detection rate
National Software Quality Experiment
Net savings
software inspections
software product engineering
Return on investment

Abstract
In order to deliver superior quality many organizations have made commitments to initiatives on the Software Engineering Institute's Capability Maturity Model, International Standards Organization (ISO) 9001, or Six Sigma. Each of these initiatives has one thing in common: the practice of Software Inspections.

Software Inspections are structured to serve the needs of quality management in verifying that a software artifact complies with its standard of excellence for software engineering. The focus is on verification, that is, on doing the job right. The Software Inspection is a formal review held at the conclusion of a life cycle activity and serves as a quality gate with an exit criteria for moving to subsequent activities. The National Software Quality Experiment (NSQE) is providing valuable insights on the practice of Software Inspections through its database of thousands of Software Inspection sessions from dozens of organizations containing tens of thousands of defects along with the pertinent information needed to pinpoint their root causes.

This model for Return on Investment bases the savings on the cost avoidance associated with detecting and correcting defects earlier rather than later in the product evolution cycle. It is defined as Net Savings divided by Detection Cost, where Net Savings is Cost Avoidance less Cost to Repair Now and Detection Cost is the cost of preparation effort and the cost of conduct effort. Savings result from early detection and correction avoiding the increased cost multiplier associated with detection and correction of defects later in the life cycle. A Major Defect that leaks from Development to Test may cost two to ten times to detect and correct. Some of these defects leak further from Test to Customer Use and may cost an additional two to ten times to detect and correct. A Minor Defect may cost an additional two to four times to correct later. The defined measurements collected in the Software Inspections Lab may be combined in complex ways to form the derived metric for return on investment. These involve additional cost multiplier, defect detection rate, cost to repair, and detection cost.

Return on Investment

Managers are interested in knowing the return on investment to be derived from software process improvement actions. The Software Inspections Process gathers some of the data needed to determine this [McGibbon 96]. Software Inspections are structured to serve the needs of quality management in verifying that a software artifact complies with its standard of excellence for software engineering. The focus is one of verification, on doing the job right. The software inspection is a formal review held at the conclusion of a life cycle activity and serves as a quality gate with an exit criteria for moving on to subsequent activities [O'Neill 01].

This model for Return on Investment bases the savings on the cost avoidance associated with detecting and correcting defects earlier rather than later in the product evolution cycle. A Major Defect that leaks from Development to Test may cost two to ten times to detect and correct. Some of these defects leak further from Test to Customer Use and may cost an additional two to ten times to detect and correct. A Minor Defect may cost an additional two to four times to correct later [O'Neill 01].

The Return on Investment for Software Inspections is defined as Net Savings divided by Detection Cost, where Net Savings is Cost Avoidance less Cost to Repair Now and Detection Cost is the cost of preparation effort and the cost of conduct effort. Savings result from early detection and correction avoiding the increased cost multiplier associated with detection and correction of defects later in the life cycle.

The defined measurements collected in the Software Inspections Lab may be combined in complex ways to form the derived metric for return on investment. These involve additional cost multiplier, defect detection rate, cost to repair, and detection cost.

Software Product Engineering Method

The values for these complex parameters revolve around the software product engineering method being practiced. Three levels of achievement of software product engineering are identified:

1. Ad hoc programming is characterized by a code and upload life cycle and a hacker coding style. This is common in low software process maturity organizations especially those facing time to market demands.
2. Structured software engineering employs structured programming, modular design, and defined programming style and pays close attention to establishing and maintaining traceability among requirements, specification, architecture, design, code, and test artifacts. This is the minimum expectation for an SEI CMM level 3 [Paulk 95].
3. Disciplined software engineering is more formal and might be patterned after Clean Room software engineering, Personal and Team Software Process, and Extreme Programming techniques [Prowell 99], [Humphrey 97], [Wells 01]. This is the expectation for an SEI CMM level 4 and 5 organization [Paulk 95].

Additional Cost Multiplier

Since savings result from early defect detection and correction avoiding the increased cost multiplier associated with detection and correction of defects later in the life cycle, the question of the cost multiplier must be answered in order to determine the return on investment. Some set

the additional cost multiplier for finding and fixing a defect detected after delivery at 100 times [Basili/Boehm 01]. Others have measured it more precisely and found it to be 10 times for each life cycle activity. IBM Rochester, winner of the Malcolm Baldrige Award, reported that defects leaking from code to test cost nine times more to detect and correct, and defects leaking from test to the field cost thirteen times more [Lindner 94].

Why is There a Multiplier?

An example may help illustrate why a leaked defect costs more. A code defect that leaks into testing may require multiple test executions to confirm the error and additional executions to obtain debug information. Once a leaked defect has been detected, the producing programmer must put aside the task at hand, refocus attention on correcting the defect and confirming the correction, and then return to the task at hand. The corrected artifact must then be reinserted into the software product engineering or product release pipeline.

What is The Multiplier?

It is reasonable to expect the additional cost multiplier to be linked to the software product engineering method practiced. Figure 1 portrays the Additional Cost Multiplier by Software Product Engineering method.

1. Ad hoc programming (AHP) is likely to experience a multiplier of 8-10 times in detecting and correcting major defects in spaghetti bowl coding lacking in order and consistency. The multiplier for minor defects is likely to be 4 times.
2. Structured software engineering (SSE) is likely to experience a multiplier of 5-7 times in detecting and correcting major defects in the production of well structured, consistently recorded components with organized relationships among modules and traceability among life cycle artifacts. The multiplier for minor defects is likely to be 3 times.
3. Disciplined software engineering (DSE) with its formal focus on quality may experience a multiplier of 2-4 times in detecting and correcting major defects. The multiplier for minor defects is likely to be 2 times.

What Effect Does the Multiplier Have?

In summary, an undetected major defect that escapes detection and leaks to the next phase of the life cycle may cost two to ten times to detect and correct. A minor defect may cost two to four times to detect and correct. The resulting Net Savings then may be up to nine times for major defects and up to three times for minor defects.

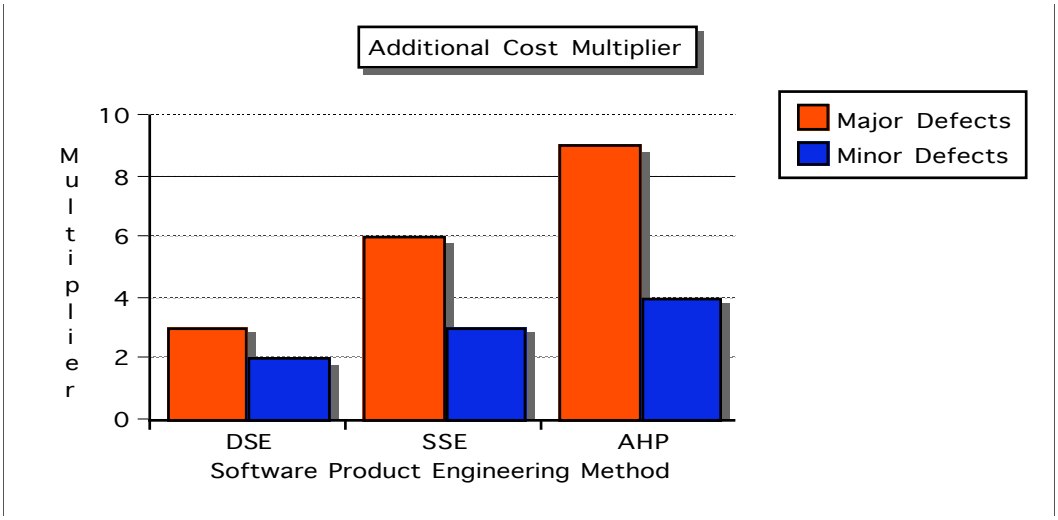

Figure 1 Additional Cost Multiplier

Defect Detection Rate

The model shown in figure 2 illustrates that defects are detected in Development (DD) and Test (TD), and defects leak from Development (DL) and Test (TL). Defect detection rate is the number of defects detected divided by the number of defects present.

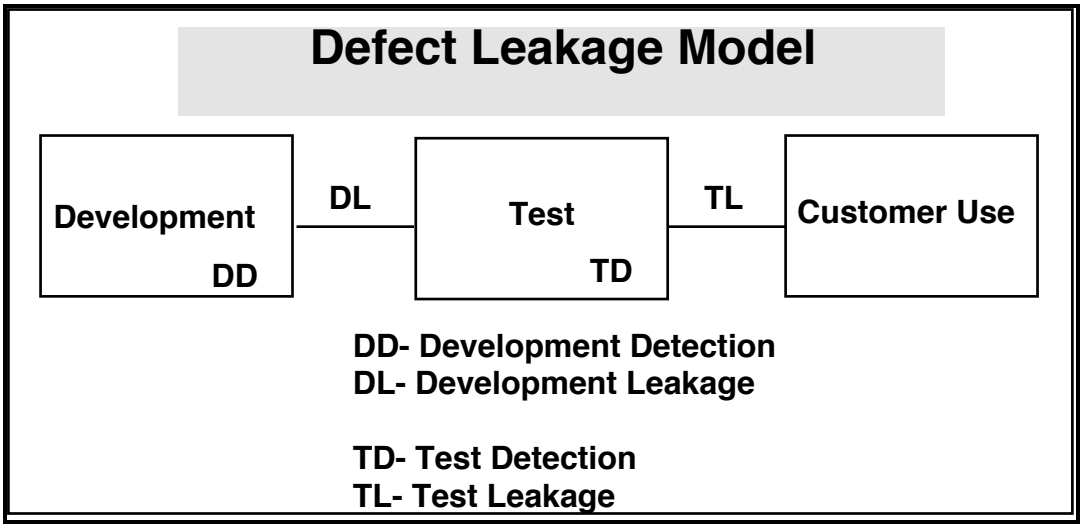

Figure 2 Defect Leakage Model

It is reasonable to expect the defect detection rate to be linked to the software product engineering method practiced including the software inspection process followed. Figures 3a-b illustrate Development Detection and Test Leakage using empirically derived values for the defect leakage model factors of each Software Product Engineering method.While the development defect detection rates are based on the results of the National Software Quality Experiment (NSQE), the expected test detection uses a notional value in order to complete the analysis.

1. Ad hoc programming is likely to experience a development defect detection rate in the range of .50 to .65. While the test leakage depends on the adequacy of the test process, ad hoc

programming is likely to experience test leakage in the range of .175 to .25 based on an expected test detection of .50.
2. Structured software engineering is likely to experience a development defect detection rate in the range of of .70 to .80 and a test leakage in the range of .1 to .15 based on an expected test detection of .50.
3. Disciplined software engineering may experience a development defect detection rate in the range of .85 to .95 and a test leakage in the range of .025 to .075 based on an expected test detection of .50.

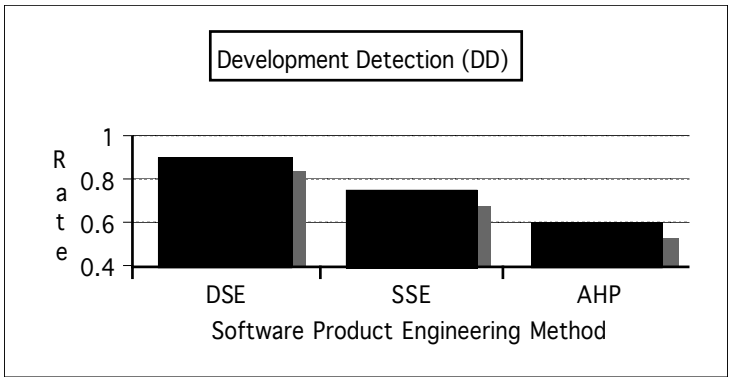

Figure 3a Development Detection

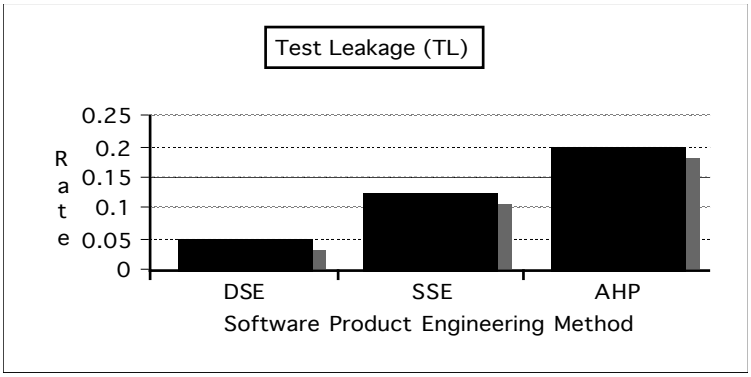

Figure 3b Test Leakage

Cost to Repair

The cost to repair a defect detected in the life cycle activity in which it was inserted depends on the software product engineering method practiced and the business environment in which it is operating. This must be supplied by the organization based on its actual cost history and the superior knowledge of its personnel.

In determining the cost to repair, the organization needs to obtain this cost by defect type. During the Software Inspection Lab session, each defect detected is assigned a defect type including interface, data, logic, I/O, performance, functionality, human factors, standards, documentation, syntax, maintainability, and other. The defect type distribution revealed by the National Software Quality Experiment is shown in figure 4 [O'Neill 95,96,98,99,00].

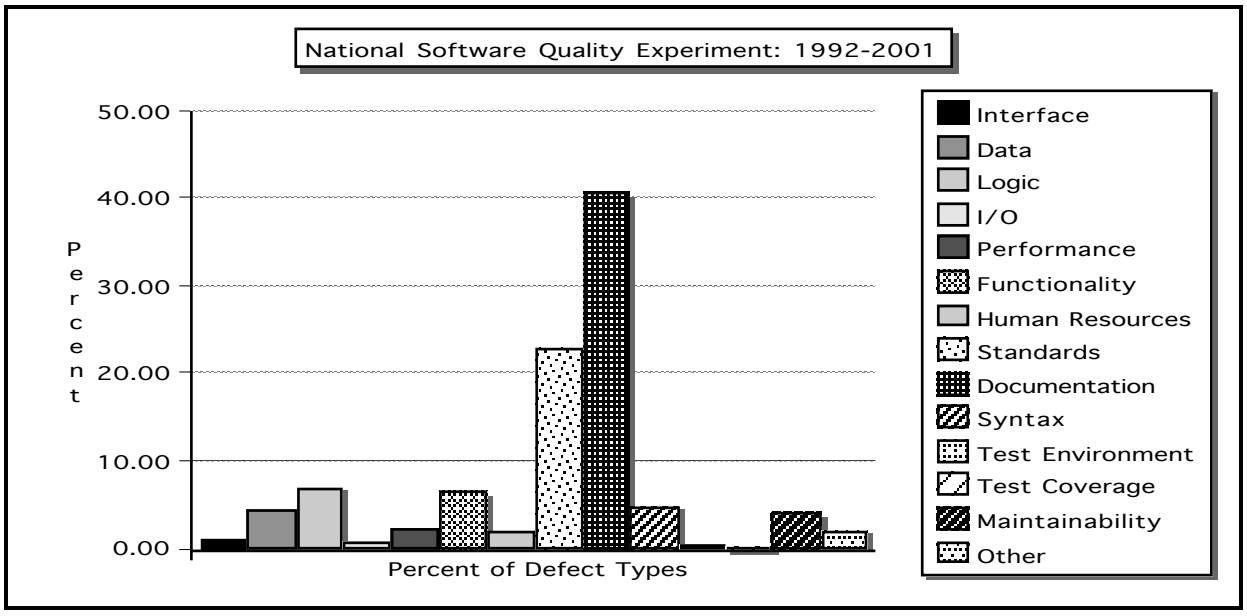

Figure 4 Defect Type Distribution

For purposes of the Software Inspections Return on Investment analysis, the cost to repair factor is included in the expression for Net Savings discussed later. For analysis here, the cost to repair is set at one hour for a major defect and one hour for a minor defect.

Defect Detection Cost

The cost of defect detection includes the preparation effort and conduct effort of the Software Inspection participants. Conduct time is the wall clock time consumed by the Software Inspection meeting. Conduct effort is conduct time times the number of participants. Factors involved in determining detection cost include the size of the artifact being inspected, the number of defects inserted, and the relationship between preparation effort and conduct effort.

It is reasonable to expect the defect detection cost to be linked to the software product engineering method practiced including the software inspection process followed. Figure 5 illustrates the defect insertion rates by Software Product Engineering method.

1. Ad hoc programming may experience a preparation effort divided by conduct effort ratio of approximately .60 in inspecting artifacts of 400-600 lines of code, as experienced by level 1 organizations in the National Software Quality Experiment [O'Neill 95,96,98,99,00]. These organizations may experience a defect insertion rate of 40-60 defects per thousand lines of code.
2. Structured software engineering may experience a preparation effort divided by conduct effort ratio of approximately .80 in inspecting artifacts of 200-400 lines of code, as experienced by level 3 organizations in the National Software Quality Experiment [O'Neill 95,96,98,99,00]. These organizations may experience a defect insertion rate of 20-30 defects per thousand lines of code.
3. Disciplined software engineering may experience a preparation effort divided by conduct effort ratio of approximately 1.0 in inspecting artifacts less than 200 lines of code. These organizations may experience a defect insertion rate of 10-15 defects per thousand lines of code.

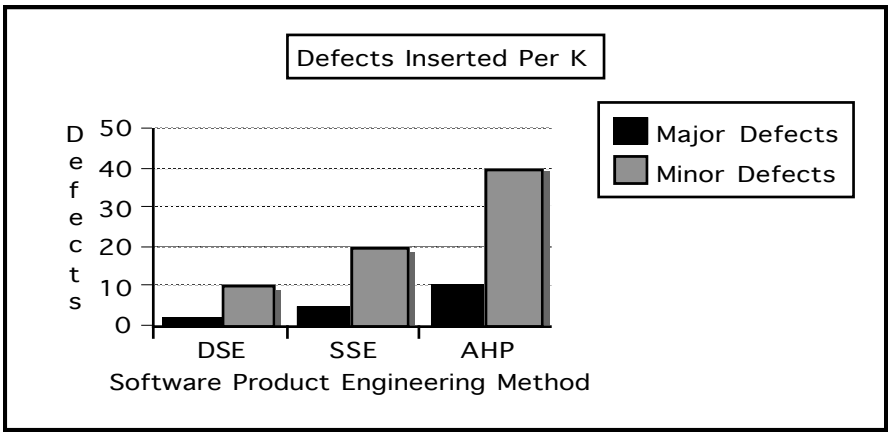

Figure 5 Defects Inserted Per Thousand Lines

Reasoning About ROI

Software Inspections Return on Investment is Net Savings divided by Detection Cost. Reasoning about Return on Investment then is assisted by evaluating the expression
[ROI:= Net Savings/Detection Cost].

Reasoning About Net Savings

Net Savings is Cost Avoidance minus Cost to Repair Now. Reasoning about Net Savings is assisted by evaluating the expression [Net Savings:= Cost Avoidance-Cost to Repair Now].

Cost Avoidance results from the avoidance of the higher costs occurring from deferred detection and corrections. The Additional Cost Multiplier comes into play. M1 is the Additional Cost to Repair Multiplier for Development to Test Major Defect Leakage. M2 is the Additional Cost to Repair for Test to Customer Use Major Defect Leakage. M3 is the Additional Cost to Repair for Minor Defect Leakage.

Reasoning about Cost Avoidance is assisted by evaluating the expression
[Cost Avoidance:= Major Defects * {(M1 * DD)+(M1 * DD) * (M2 * TL)*C1}+Minor Defects * M3].

The Cost to Repair Now, simply the cost of defect correction, is subtracted from Cost Avoidance to yield Net Savings. Hence reasoning about Net Savings is assisted by evaluating the expression
[Net Savings:= Major Defects * {(M1 * DD)+(M1 * DD) * (M2 * TL)*C1-C1}+Minor Defects * (M3-C2)].
Where:

M1: (2-10)	Additional Cost to Repair Multiplier for Development to Test Major Defect Leakage
M2: (2-10)	Additional Cost to Repair Multiplier for Test Customer Use Major Defect Leakage
M3: (2-4)	Additional Cost to Repair for Minor Defect Leakage
DD: (.5-.95)	Defect Detection Rate for Development to Test
TL: (.05-.5)	Test Leakage Rate for Test to Customer Use
C1:	Average Cost to Repair Major Defect

C2: Average Cost to Repair Minor Defect

Reasoning About Detection Cost

Detection Cost is Preparation Effort plus Conduct Effort. Reasoning about Detection Cost is assisted by evaluating the expression
[Detection Cost:= Preparation Effort + Conduct Effort].

Preparation Effort is the total Minutes of Preparation Effort. Conduct Effort is the Minutes of Conduct Time multiplied by the number of participants. Substituting, the resulting expression is [Detection Cost:= {Minutes of Preparation Effort + (Minutes of Conduct Time * Participants)}/60].

Where:
Participants: (4-6) Number of participants
60 minutes per hour

A Worked Example

The Return on Investment is determined by using the expression for Net Savings specified above and setting the parameters for Cost to Repair Multiplier, Defect Detection, and Defect Leakage. For example, to determine the expression for ROI to be used in a project spreadsheet, the following example is offered:

1. Setting the parameters:
M1: 5 M2: 10 M3: 2 DD: .6 TL: .25 C1: 1 C2: 1

2. Using the expression:
[Net Savings:= Major Defects * {(M1 * DD)+(M1 * DD) * (M2 * TL)*C1-C1}+ Minor Defects * (M3-C2)]

3. Substituting for the values of the worked example:
[Net Savings:= Major Defects * {(5 * .6)+(5 * .6) * (10 * .25)*1 -1}+Minor Defects * (2-1)]

4. The following expression for Net Savings results:
[Net Savings:= 9.5 * Major Defects+Minor Defects]

The result of the worked example is a simplified expression for Net Savings of the type used to derive the Return on Investment metric in the National Software Quality Experiment (NSQE). Figure 6 illustrates the range of practice for Return on Investment.

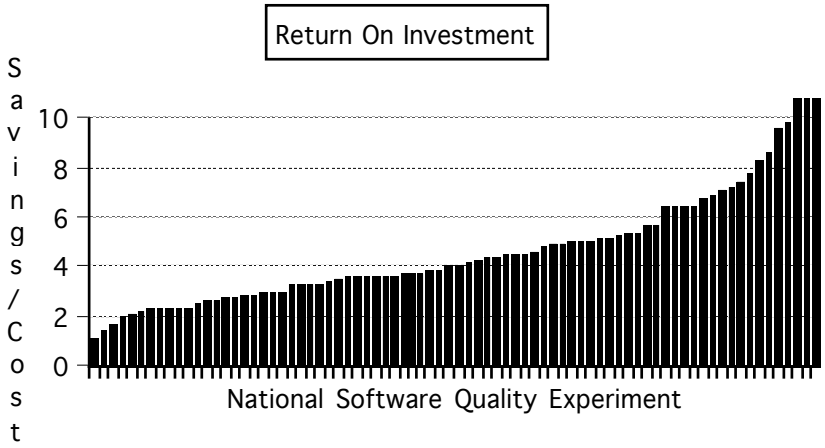

Figure 6 Return on Investment

Selecting Parameter Values

Where an organization possesses superior knowledge of its software operation, it should utilize the parameter values that best reflect this understanding. Candidate parameter values for each software product engineering method are shown below for Disciplined Software Engineering (DSE), Structured Software Engineering (SSE), and Ad Hoc Programming (AHP).

	M1	M2	M3	Major Per K	Minor Per K	DD	TL	Prep Min	Conduct Min	Partici
DSE	2-4	2-4	2	2.5	10	.95-.85	.025-.075	500	120	4
SEE	5-7	5-7	3	5	20	.70-.80	.075-.150	400	120	4
AHP	8-10	8-10	4	10	40	.50-.65	.175-.250	300	120	4

Computing ROI

Software process improvement goals involve both cost and quality. The achievement of these goals varies according to the software product engineering method practiced, and these variations are illustrated in the application of the selected parameter values. Ad hoc programming practitioners derive a substantial net savings and return on investment but a high incidence of defect leakage into customer use. Structured software engineering practitioners experience an attractive net savings and return on investment and a reduced defect leakage in to customer use. Disciplined software engineering practitioners barely recoup the investment but achieve a very low incidence of defect leakage into customer use.

	DD	M1	TL	M2	M3	Net Savings	Detection Cost	ROI	Leaks Per K
DSE									
Disciplined	.95	2	.025	2	2	12.49	16.33	0.76	.3125
Software	.90	3	.050	3	2	15.26	16.33	0.93	.6250
Engineering	.85	4	.075	4	2	18.55	16.33	1.14	.9375
SSE									
Structured									
Software	.80	5	.100	5	3	65.00	14.67	4.43	2.500
Engineering	.75	6	.125	6	3	74.38	14.67	5.07	3.125
	.70	7	.150	7	3	85.23	14.67	5.81	3.750
AHP									
Ad Hoc	.65	8	.175	8	4	234.80	13.00	18.06	8.750
Programming	.60	9	.200	9	4	261.20	13.00	20.09	10.00
	.55	10	.225	10	4	288.75	13.00	22.21	11.25
	.50	10	.250	10	4	285.00	13.00	21.92	12.50

Transition From Cost to Quality

In using Software Inspections, the goals vary with the software product engineering method used, transitioning from cost to quality.

By necessity, the focus of ad hoc programming practitioners is on reducing cost by detecting as many defects as possible. With 40-60 defects inserted, a defect detection rate of .50-.65, and an additional cost multiplier of 8-10, the result is a Net Savings of 234.80 to 285 labor hours and a defect leakage expectation of 8.75-12.50 per thousand lines of code, numbers that promote a focus on cost. For this group, finding defects is like finding free money, and there are always more defects to find; however, managers struggle to meet cost and schedule commitments.

Structured software engineering focus is split between reducing cost and improving quality. With 20-30 defects inserted, a defect detection rate of .70-.80, and an additional cost multiplier of 5-7, the result is a a Net Savings of 65.00-85.23 labor hours and a defect leakage expectation of 2.5-3.75 per thousand lines of code, numbers that promote an attraction to both goals. For this group, there is constant dithering between between cost and schedule.

Without question, the focus of disciplined software engineering practitioners is on eliminating every possible defect even if defect detection costs exceed net savings and the return on investment falls below the break even point. With 10-15 defects inserted, a defect detection rate of .85-.95, and an additional cost multiplier of 2-4, the result is a Net Savings of 12.49-18.55 labor hours and a defect leakage expectation of .3125-.9375 per thousand lines of code, numbers that promote a focus on quality. For this group, every practitioner is riveted on achieving perfection.

To Compute Your Return on Investment

When an organization has superior knowledge of the parameter values for software inspections return on investment, it is able to derive its own ROI metric. To perform this computation,

simply visit the tool at http://members.aol.com/ONeillDon/nsqe-roi.html

Bibliography

[Basili/Boehm 01] Basili, Vic, Barry Boehm, "Top Ten Defect Reduction List", IEEE Software, January 2001

[Humphrey 97] Humphrey, Watts, "Introduction to the Personal Software Process", Addison-Wesley, Reading, Massachusetts, 1997

[Lindner 94] Lindner, Richard J. & Tudahl, D. "Software Development at a Baldrige Winner," Proceedings of ELECTRO `94, Boston, Massachusetts, May 12, 1994, pp. 167-180.

[McGibbon 96] McGibbon, T., "A Business Case for Software Process Improvement", Rome Laboratory DACS Report, 30 September 1996.

[O'Neill 98] O'Neill, Don, "National Software Quality Experiment: A Lesson in Measurement 1992-1997", CrossTalk, The Journal of Defense Software Engineering, Vol. 11 No. 12, Web Addition, December 1998.

[O'Neill 99] O'Neill, Don, "National Software Quality Experiment: A Lesson in Measurement 1992-1997", First Annual International Software Assurance Certification Conference, Chantilly, Virginia, 1 March 1999, pp. 1-14.

[O'Neill 95,96,00] O'Neill, Don, "National Software Quality Experiment: Results 1992-1999", Software Technology Conference, Salt Lake City, 1995, 1996, and 2000

[O'Neill 01] O'Neill, Don, "Peer Reviews", Encyclopedia of Software Engineering, Wiley Publishing, Inc., January 2002

[Paulk 95] Paulk, Mark C., "The Capability Maturity Model: Guidelines for Improving the Software Process", Addison-Wesley Publishing Company, 1995, pp. 270-276.

[Prowell 99] Prowell, Stacy J., Carmen J.Trammell, Richard C. Linger, Jesse H. Poore, "Cleanroom Software Engineering: Technology and Process", Addison-Wesley Longman, Inc., 1999, page 17, 33-90.

[Wells 01] Wells, J. Donovan, http:///www.extreme programming.org

3648 words

Don O'Neill

Don O'Neill is a seasoned software engineering manager and technologist currently serving as an independent consultant. Following his twenty-seven year career with IBM's Federal Systems Division, Mr. O'Neill completed a three year residency at Carnegie Mellon University's Software Engineering Institute (SEI) under IBM's Technical Academic Career Program.

As an independent consultant, Mr. O'Neill conducts defined programs for managing strategic software improvement. These include implementing an organizational Software Inspections Process, directing the National Software Quality Experiment, implementing Software Risk Management on the project, conducting the Project Suite Key Process Area Defined Program, and conducting Global Software Competitiveness Assessments.

He is a founding member of the Washington DC Software Process Improvement Network (SPIN) and the National Software Council (NSC) and serves as the Executive Vice President of the Center for National Software Studies (CNSS). He is a collaborator with the Center for Empirically-based Software Engineering (CeBASE). Mr. O'Neill has a Bachelor of Science degree in mathematics from Dickinson College in Carlisle, Pennsylvania.

Contact Information

Don O'Neill
Independent Consultant
9305 Kobe Way
Montgomery Village, Maryland 20886

email: ONeillDon@aol.com
Phone: (301) 990-0377
http://members.aol.com/ONeillDon/index.html