The Development of Zero Defect Computer Software

G. Gordon Schulmeyer, CDP

1 December 1991

Submitted for
The Shingo Prize for Excellence in Manufacturing Research

Prepared for
PYXIS Systems International, Incorporated

Abstract

This paper applies the principles of Shigeo Shingo from his book, *Zero Quality Control: Source Inspection and the Poka-yoke System*, to the development of computer software.

Process in relation to computer software development is examined with special emphasis given to Michael Fagan's inspection process and Shigeo Shingo's cycle for managing errors and defects.

A computer software development chart is given, explaining its reliance on Shigeo Shingo's source inspection concepts. The idea that every computer software developer is responsible for his or her own quality through source inspections is explained. The use of source inspections by developers to hand-off products to internal customers is compared to the successive inspections required by those internal customers before they accept the product.

The concept of Shigeo Shingo called poka-yoke is extended to the computer software development arena through the use of software tools providing automated mistake proofing to the development process.

The paper concludes with a review of the management and worker responsibilities inherent in a successful computer software development process.

Introduction

This paper lays a framework for the development of zero defect computer software by applying the manufacturing principles of Shigeo Shingo to computer software development.

First, why the only acceptable quality level for computer software development is at the zero defects level is explored. Here, a software defect is any software failure delivered to the ultimate customer.

A number of models and issues concerning the computer software process are next investigated highlighting inspection points.

The synergism of the important concepts in Shigeo Shingo's book, *Zero Quality Control: Source Inspections and the Poka-yoke System* (pronounced POH-kah YOH-kay) and the procedure charting of value analysis leads to a guide toward the achievement of zero defect computer software.

With the outline provided by the computer software development process chart (appendix of the *Zero Defect Software* book), the key issues of importance to its successful implementation are discussed. The customer is discussed in terms of the next person in line in the development process.

Errors are human and will always be made. The secret to successful zero defect computer software is to isolate the errors that humans make along the way and remove them. Source inspections and successive inspections isolate errors manually, and poka-yoke techniques help prevent errors with an automated device, so no software defects get delivered.

The responsibilities of both the individuals (workers) and the managers in the software development process are next examined. This is because it is only through people that the achievement of the zero defect computer software goal may be obtained.

ZERO DEFECTS OR ACCEPTABLE QUALITY LEVEL

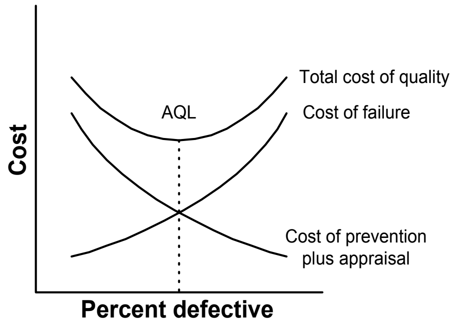

Figure 1: Acceptable Quality Level (Burrill & Ellsworth, 1982, p. 50)

There are conflicting schools of thought about quality. One says only zero defects is acceptable. The other says set an acceptable quality level (AQL) because the cost and effort to achieve zero defects is not worthwhile. Figure 1 gives an AQL chart where producing above AQL quality is out of control, but producing below AQL is too expensive to be practical.

Burrill and Ellsworth in *Quality Data Processing* point out that Figure 1 does not adequately represent the cost of computer software systems (critical products) failure. A software malfunction can generate costs that are extremely large compared with the cost of system development. This means that the cost of failure rises steeply as the percent of defectives increase.

Second, consider the cost of prevention plus appraisal. The traditional view is that this cost must become almost infinite to achieve zero defects. But this is not true for software systems. Examples of no defect software in the first year of operation show that zero defects can be achieved at a cost of prevention plus appraisal that is finite and reasonable. Also, the increase in this cost is not excessive as the percent of defects trends to zero.

For failure costs and prevention plus appraisal costs as shown in Figure 2, the total cost of failure decreases steadily as the percent defective trends to zero. So the minimum total cost of failure is at the point of zero defects. For these curves, the AQL is zero defects. (Burrill & Ellsworth, 1982, pp. 49-51)

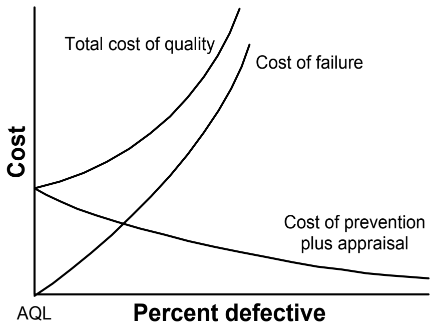

Figure 2: AQL is Zero Defects (Burrill & Ellsworth, 1982, p. 52)

"A process may be described as a set of operations occurring in a definite sequence that operates on a given input and converts it to some desired output. (Fagan, 1976, p. 125) For software development, explicit requirement statements are required as input. The series of processing operations that act on this input must be placed in the correct sequence with one another, the output of each operation satisfying the input needs of the next operation. The output of the final operation is the explicitly required output in the form of a verified program. Thus, the objective of each processing operation is to received a defined input and to produce a definite output that satisfies a specific set of exit criteria. A well-formed process can be thought of as a continuum of processing during which sequential sets of exit criteria are satisfied, the last set in the entire series requiring a well-defined end product. (Fagan, 1976, p. 125)

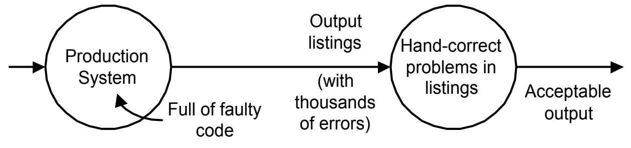

Figure 3: Correct the Program (DeMarco, 1982, p. 206)

Would you calculate every week the results that go on your listing: i.e., correct the listing every week (Figure 3). (DeMarco, 1982, p. 206) It is a reasonable assumption that no one is so ignorant that they would not correct the program once, instead of continually correcting the results.

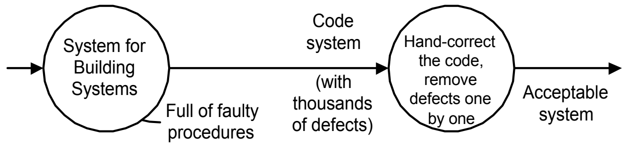
System for Building Systems
Full of faulty procedures
Code system
(with thousands of defects)
Hand-correct the code, remove defects one by one
Acceptable system

Figure 4: Correct the System (DeMarco, 1982, p. 206)

Figure 4 shows the same situation with some labels changed. It shows a staff repeatedly correcting defective products of a system rather than correcting the system that produced those defective products. There are billions of dollars spent correcting system defects without realizing the wrong system is being corrected. The system for building systems should be corrected.

When a software developer says. "This is the problem" while pointing to listings, he or she is really only pointing out the symptom, not the problem. The real problem is a defective development procedure that allows defective code to be put there. We do not remove defects from the system for building systems, because we never go looking for them. It is better to correct the system for building systems so that it does not insert defects in the first place. (DeMarco, 1982, p. 205-207)

Michael Fagan's landmark work on the software development process using inspections to improve quality provided several charts showing that fewer errors resulted the closer to the work the inspection was held. Figure 5 shows an example of how design inspections, I_1, code inspections, I_2, and test inspections, I_3, resulted in 38 percent fewer errors per thousand lines of code (KLOC). Figure 6 compares the old approach with the one Michael Fagan proposed showing a reduction in overall schedule using requirements inspection, I_0, and I_1, and I_2 inspections embedded in the process.

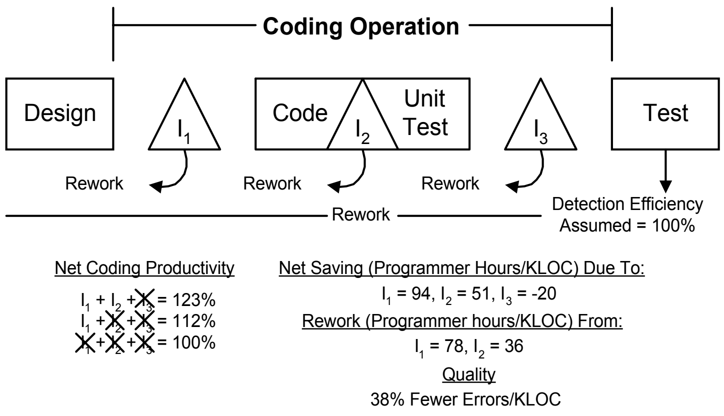
Coding Operation
Design
I_1
Code
I_2
Unit Test
I_3
Test
Rework
Rework
Rework
Rework
Detection Efficiency Assumed = 100%
Net Coding Productivity
$I_1 + I_2 + \cancel{I_3}$ = 123%
$I_1 + \cancel{I_2} + \cancel{I_3}$ = 112%
$\cancel{I_1} + \cancel{I_2} + \cancel{I_3}$ = 100%
Net Saving (Programmer Hours/KLOC) Due To:
I_1 = 94, I_2 = 51, I_3 = -20
Rework (Programmer hours/KLOC) From:
I_1 = 78, I_2 = 36
Quality
38% Fewer Errors/KLOC

Figure 5: Inspection Model with Statistics from Sample Problem (Fagan, 1976, p. 127)

Shigeo Shingo (1986/1985) has devised a process cycle for the managing of errors and defects (Figure 7). The check and feedback portion is analogous to inspections I_0, I_1, I_2, and I_3 shown in Figure 5 and 6. Note that the short route similar to Fagan's approach gets closer to the worker. Getting the worker involved in the process of quality improvement of their own process is one of the driving forces behind the computer software development process chart discussed next.

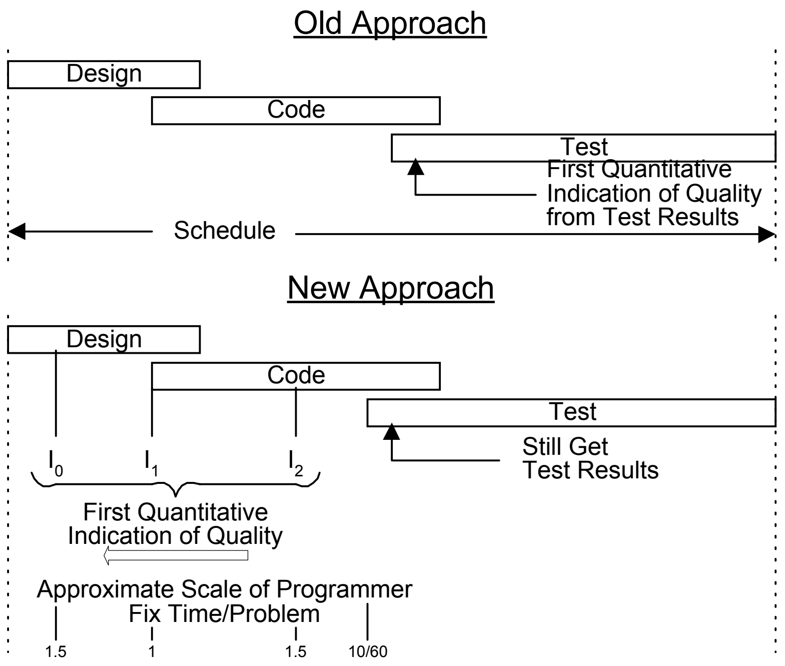
Old Approach
Design
Code
Test
First Quantitative Indication of Quality from Test Results
Schedule
New Approach
Design
Code
Test
Still Get Test Results
I_0
I_1
I_2
First Quantitative Indication of Quality
Approximate Scale of Programmer Fix Time/Problem
1.5
1
1.5
10/60

Figure 6: Effect of Inspection on Process Management (Fagan, 1976, p. 144)

Computer Software Development Process Chart

Shigeo Shingo's book, *Zero Quality Control: Source Inspections and the Poka-yoke System* convinced this author that Shingo's fundamental manufacturing ideas to achieve zero defects could be applied to software development. Shingo's approach, combined with concepts of Value Analysis, facilitated generating the Computer Software Development Process Chart. (See Appendix to *Zero Defect Software* book).

The Computer Software Development Process Chart starts with a modified collection of symbols usually used in value analysis to show paper flow in a company. With the use of this type of flow analysis, methods can be found to improve a process.

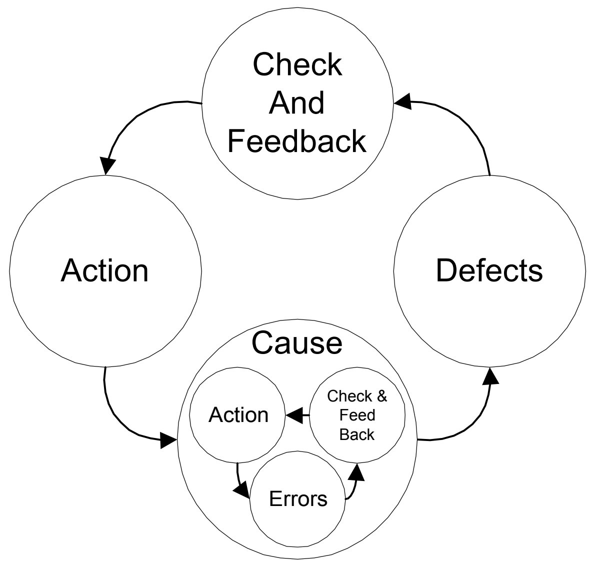

Figure 7: Cycle for Managing Errors and Defects (Shingo, 1986/1985, p. 53)

The entire computer software development chart based upon the value analysis symbols is in the appendix of the *Zero Defect Software* book by the author. It is derived primarily from software development methods as outlined in DOD-STD-2167A, *Defense System Software Development* (1988).

Every product, whether it be a document or work product of computer software development, has an informal review which is the self-checking by the worker who produced it to check its integrity. This also takes place whenever a work product is updated, which happens frequently during software development. This is called a self-check inspection, to be discussed in more detail later.

If this work product is to be handed off to another, this is the place to get that other person -the internal customer -into the process. The receiver has a vested interest in what he or she is going to have to work with and so will be critically sure that this a good product. This is called a successive check inspection system also discussed later.

After the generation or update of each work product, a quality review of the work product takes place. Of course, this is after the worker has reviewed his or her own work. The quality review requires independence of judgment to best achieve its results.

Every formal customer review, such as a critical design review, has an internal practice run to ensure readiness for the real thing. This dry run ensures that the customer sees only the correct results, not intermediate errors that previously occurred. This is work that already has been through multiple reviews. Poka-yoke techniques do not show up in this portion of the chart. This does not imply that they should not be used here. A discussion of their usage and how they could be used in the process is in a later section.

THE CUSTOMER

Customer satisfaction must be considered from two viewpoints. First is the ultimate customer who is paying for the product or service. The second customer consideration relates to the next person in the chain of operations within an organization. Both must be satisfied for successful quality improvement.

Three major points to improve employee involvement with the ultimate customer are that every contact should be a quality interface, responsiveness to the customer is important, and involving customers in activities (for example, design reviews) related to their work helps to build a team approach that results in openness and trust.

The concepts applied to the ultimate customer are applicable to the internal customer. Meeting the internal customer's requirements completely and in a timely manner usually contributes to the ultimate customer's satisfaction. Infrequently there is a conflict and, when that occurs, the ultimate customer must come first.

Everyone has customers and success comes from satisfying those customers. This is achieved by meeting customer expectations, which, of course, is meeting requirements. Customer satisfaction is an integral part of any quality improvement process. Goals, objectives, and action plans for internal and ultimate customer satisfaction must be established, measured, and managed. (Cooper, 1987, pp. 13-16)

In order to be truly customer-oriented, every worker must find out what the next person in line needs to do a good job and then provide it. In a well-managed business, each employee considers the next person in line as the customer and works to help him to do a better job. (Tribus, 1987, pp. 24-27) This is a vital concept in the computer software development process chart that can lead to zero defect software.

In "The Push for Quality" (1987, June 8) from *Business Week* magazine this concept was explained as follows: that every person on a production line is the customer of the proceeding operation (called the internal customer above), so each worker's goal is to make sure that the quality of his work meets the requirements of the next person. When that happens throughout the organization, the satisfaction of the ultimate customer should be assured. This concept was revolutionary to some workers who realized that they have a customer.

ISOLATING THE ERROR

Mr. Shingo tells us that since defects are generated during the process, you only discover those defects by inspecting goods at the end of the process. Adding inspection workers is pointless because you will not reduce defects without using processing methods that prevent defects from occurring in the first place.

It is an unalterable fact that processing produces defects and all that inspections can do is find those defects. So approaching the problem only at the final inspection stage is meaningless. Defects will not be reduced merely by making improvements at the final inspection stage, although such improvements may eliminate defects in delivered goods.

The most fundamental concept is to recognize that defects are generated by work and all final inspections -judgmental inspections - do is to discover those defects. Zero defects can never be achieved if this concept is forgotten. (Shingo, 1986, pp. 35-39)

In terms of its effect on defect density, software testing borders on the irrelevant. In the United States, the range of defects per thousand lines of executable code is from 0.016 to 60, a factor of nearly 4000. The way to make a drastic improvement in the quality of code that comes out of the testing process is to make a drastic improvement in the quality of code that goes into the testing process. (DeMarco, 1982, pp. 216-218) One way is to use computer-based systems to analyze product quality while the design detail are merely images and symbols on a computer screen. Do this because 80 percent of all product defects get "designed in". (Kotelly, 1985, p.9)

Shigeo Shingo (1986/1985) describes three major inspection methods: (1) judgment inspections that discover defects. (2) informative inspections that reduce defects, and (3) source inspections that eliminate defects. Judgment inspections were just discussed above.

In an informative inspection, information of a defect occurring is fed back to the specific work process, which then corrects the process. Consequently, adopting informative inspections regularly should gradually reduce production defect rates. There are three categories of informative inspections:

- o Statistical Quality Control Systems (SQCS)
- o Successive Check Systems (SuCS)
- o Self-Check Systems (SeCS)

Statistical Quality Control Systems (SQCS) include the notion of informative inspections and use statistically based control charts. SQC systems use statistics to set control limits that distinguish between normal and abnormal situations. The essential condition identifying a method of inspection as an SQC method is the use of statistical principles. (Shingo, 1986/1985 pp. 58-59)

SQC systems suffer from two shortcomings:

1. Sampling is used. Would not it be better to use 100 percent inspections to find all abnormalities. One-hundred percent inspections, however, are expensive and time consuming. If low-cost, l00-percent inspections could be devised, would not they be preferable? This leads to poka-yoke devices.
2. SQC methods are too slow to be fully effective concerning feedback and corrective action.

The best way to speed up feedback and corrective action would be to have the worker who finds any abnormality carry out 100-percent inspections and immediately take corrective action. But objectivity is essential to the performance of inspections, and that is why inspections have been carried out by independent inspectors. An inspection can be carried out by any worker other than the one who did the processing. If this task is given to the nearest person, then one could have a successive check system of the following sort:

1. When A completes processing, he passes it on to B for the next process.
2. B first inspects the item processed by A and then carries out the processing assigned to him. Then B passes the item to C.
3. C first inspects the item processed by B and then carries out the processing assigned to him. Then C passes the item to D.
4. Similarly, each successive worker inspects items from the previous process.
5. If a defect is discovered in an item from the previous process, the defective item is immediately passed back to the earlier process. There, the item is verified and the error corrected. Action is taken to prevent the occurrence of subsequent errors.

Successive check systems represent an advance over control chart systems because it makes it possible to conduct 100-percent inspections, performed by people other than the workers involved in the processing. (Shingo, 1986/1985, pp. 67-69) This does not imply a look at the entire output data population for software, but a look at the computer programs and associated documentation.

The nature of informative inspections remains such that rapid feedback and swift action are desirable, and it would be ideal to have the actual worker involved conduct 100-percent inspections to check for defects. However, there are two flaws to be reckoned with: workers are liable to make compromises when inspecting items that they themselves have worked on and they occasionally forget to perform checks on their own.

If these flaws could be overcome, then a self-check system would be superior to a successive check system. In cases where physical, rather than sensory, inspections are possible, poka-yoke devices can be installed within the process boundaries, so that when abnormalities occur, the information is immediately fed back to the worker involved. Because abnormalities are discovered within the processes where they occur rather than at subsequent processes, instant corrective action is possible. So, a self-check system is a higher order approach than the successive check system to cut defects even further. (Shingo, 1986/1985, pp. 77)

Shigeo Shingo (1986/1985) further explains that source inspections are inspection methods that are based on discovering errors in conditions that give rise to defects and performing feedback and action at the error stage so as to keep those errors from turning into defects, rather than stimulating feedback and action in response to defects. This is graphically portrayed in Figure 7.

Many people maintain that it is impossible to eliminate defects from tasks performed by humans. This stems from the failure to make a clear separation between errors and defects. Defects (delivered to the customer) arise because errors are made; the two have a cause-and-effect relationship.

It is impossible to prevent all errors from occurring in any task performed by humans. Inadvertent errors are possible and inevitable.
However, if feedback and corrective action are optimized, errors will not turn into defects. The principle feature of source inspections eliminates defects by distinguishing between errors and defects; i.e., between causes and effects. (Shingo, 1986/1985, pp. 82-85)

Table 1. Example Software Tool Types and Categories (Cavano & LaMonica, 1987, p. 29)

General Support:	Prototyping:
• Text Editor	• Window Prototyper
• Document Formatter	Verification:
• Command-Language Editor	• Data Tracer
• Code Auditor	• Static Analyzer
• Electronic Mail	• Dataflow Analyzer
Requirements Analysis:	• Interface Checker
• Requirements Generator	• Quality Analyzer
• Requirements Documentor	Configuration Management:
• Consistency Analyzer	• Software Manager
Design:	• Documentation Manager
• Design Generator	• Test-Data/Results Manager
• Design Documentor	• Change-Effect Manager
• Consistency Checker	Project Management:
Coding:	• Project Planner
• Language-Sensitive Editor	• Project Tracker/Reporter
• Assembler	• Problem-Report Processor
• Compiler	• Change-Request
• Linker	Environment Management:
• Debugger	• Method Script Editor
• Assertion Translator	• Menu Editor
Testing:	• Keypad Editor
• Instrumentor	• Command-Procedure Editor
• Postexecution Analyzer	• Global-Command Setup
• Test-Summary Reporter	• Tool Installer/Deleter
• Test Manager	
• Simulator	

Shigeo Shingo used poka-yoke (mistake-proofing) for devices that serve to prevent (or "proof" in Japanese, yoke) the sort of inadvertent mistake (poke in Japanese) that anyone could make. Poka-yoke systems can be combined with successive checks or self-checks and can fulfill the needs of those techniques by providing 100-percent inspections and prompt feedback and action. Successive checks and self-checks function only as information inspections in which feedback and action take place after a defect has occurred. In cases where repairs can be made, it looks as though no defects occurred, but these inspection methods alone are inherently unable to attain zero defects. (Shingo, 1986/1985, p. 92)

For software an example of poka-yoke is the Software Life-Cycle Support Environment (SLCSE). Rather than supporting a single development methodology, SLCSE uses a modifiable tool-based approach where an almost unlimited number of tools can be integrated to support various methodologies. Off-the-shelf or custom tools may be used so that changes in the modes or the development paradigm may be supported. Table 1 shows initial tool categories and representative generic tool types for each category that will be supported by SLCSE. The tool categories reflect life-cycle phases or activities that span the entire cycle. (Cavano & LaMonica, 1987, p. 29) These tools are the equivalent of poka-yoke techniques for the computer software development process.

Source inspections and poka-yoke measures must be combined to eliminate defects. The combination of source inspections and poka-yoke devices make zero quality control systems possible.
This same combination applies to the computer software development process chart. One must never forget, finally, that the poka-yoke system refers to a means, not an end. (Shingo, 1986/1985, pp. 92,93)

RESPONSIBILITY FOR ZERO DEFECT COMPUTER SOFTWARE

Success in quality -conformance to requirements -is a management issue. Management must recognize the weaknesses of the organization and see that quality applied correctly will eliminate them. Management establishes the organizational purpose, makes measurable objectives, and takes action required to meet the objectives.
Management sets the tone for the people in the organization. If people perceive management indifference to quality, indifference will permeate the organization. On the other hand, when management is perceived to have a quality commitment, quality will permeate the organization. (Cooper, 1987, p. 6)

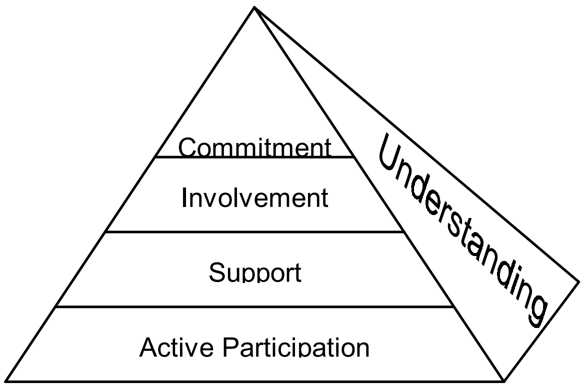

Figure 8: Management Quality Involvement (Cooper, 1987, p. 18)

To achieve this commitment, management must understand quality and be involved in the process with active participation (Figure 8). Each person in the organization must see participation by the level above him.

Management shows its commitment to quality by establishing a quality policy. The policy should set the expectations for quality and should apply to all departments and individuals in the organization. The policy needs to be simple, direct, and concise. With proper management attention to establish the policy as the basis for quality performance, it becomes a rallying point for people to determine actions and priority.

Management must eliminate all temptations to compromise conformance to requirements. It only takes one compromise to create doubt - forget that software unit test, ship that software design specification without software quality review. Once there is doubt, long-term consistent behavior on the part of management is the only way to diffuse the doubt. Without management buying into the zero defect software method it cannot succeed. Too often the cry has been heard "zero defect software is impossible", but management must provide the leadership to set the tone that zero defect software is possible or it will never be aimed for and achieved.

Attention has been focused on management's responsibilities, it is also appropriate to focus some attention on worker's responsibilities. The differences among computer software workers as shown in Figure 9 is enormous. Expect variations on the order of 10 to 1 in performance in all but the smallest project team. These variations are not caused solely by the extreme cases; they apply across the entire spectrum of workers. Even if your workers are not at the extremes, you will see differences of three or four to one in your computer software project teams.

Poor performances are responsible for an order of magnitude more defects than the best performer; lower-average performers are responsible for twice as many as upper-average. There are people on almost all projects who insert spoilage that exceeds the value of their production. Taking a poor performer off your team can often be more productive than adding a good one. In a software project team of 10, there are probably 3 people who produce enough defects to make them net negative producers. The probability that there is not even 1 negative producer out of 10 is negligible. A high defect producing team (above 30 defects/thousand lines of executable code) may have fully half in the negative production category.

There are two reasons why not much is done about negative producers: defect-prone people do not appear to be bad developers and the idea of measuring the individual is slightly repugnant to most people.

When we measure and allocate defects fairly and when we make the idea palatable to the people affected, we will find the worker who is defect prone. When we find defect-prone workers, it does not mean they should be fired, but only that they should not be allowed to write code. If A writes nearly defect-free products and B excels at defect detection and diagnosis, they assign them accordingly; let A write all the code and B do all the testing. It is unconscionable to ignore the differences between A and B, assign them correctly and the differences in project results are quite significant.

Switching roles has been shown to improve quality by more than 75 percent. Making these changes illustrates the difference in cost between removing defects ($18,000) and abstaining from defects ($0). There were 36 more defects inserted when A and B were assigned without regards to B's defect production.

If one collects accurate defect rates and efficiencies without upsetting the people involved, note the results achieved. Not only has quality improved, but each person's value to the project has been increased. No one need be embarrassed to have been assigned tasks that make optimal use of individual strengths. Even worker B, temporarily chastened to learn that all his or her testing and diagnostic skill barely made up for a high defect insertion rate, will know (once they have been assigned properly) that his or her prior low value to the project was due to management failure, rather than worker B failure. These differences in employee's value to a project often comes from the team itself. In fact, it is better to have this team participate then have management dictate.

A most important function of management is defaulted if the difference between people are ignored and they are assigned homogeneously. Measuring defect rates will prove that 25 percent or more of the staff should not be coding at all, and that some of the best coders should not be allowed to test. (DeMarco, 1982, pp. 207-210)

CONCLUSION

Using the computer software development process chart as a guide with Shigeo Shingo's source inspections and automated tools (poka-yoke), the development of zero defect computer software is defined. It has been successfully demonstrated in manufacturing by Shigeo Shingo. Now is the time for similar success in software development.

The payoff in software development is very high indeed. William Mandeville of the Carman Group, Inc. relates that 30% to 50% of product life cycle costs are wasted as the costs of poor quality. However, quality improvements of up to 75% of that loss are attainable. (Mandeville, 1989, p. 6)

REFERENCES

Burrill, C. & Ellsworth, L. (1982). *Quality data processing*. Tenafly, NJ.: Burrill & Ellsworth Associates.

Cavano, J. P. & LaMonica, F. S. (1987, September). “Quality assurance in future development environments,” *IEEE Software*, pp. 26-34.

Cooper, A. D. (1987). *The journey toward managing quality improvement*. Orlando: Westinghouse Electric.

DeMarco, T. (1982). *Controlling software projects*. New York: Yourdon.

Fagan, M. E., (1976, June). “Design and code inspections and process control in the development of programs,” *IBM-TR-00.73*, pp. 110-135.

Department of Defense. (1988). *Defense system software development* (DoD-STD-2167A). Washington, DC: SPAWAR -3212.

Graham, B. S., Jr. (1977). *Procedure charting*. Unpublished manuscript prepared for Ben S. Graham Conference.

Kotelly, G. V. (1985, December). “Competitiveness = quality,” *Mini-Micro Systems*, p. 9.

Mandeville, W. A. (1989, February). “Defects and software quality costs measurements,” *Proceedings of Fifth Annual NSIA Joint Conference*.

Shingo, S. (1986). *Zero quality control: source inspection and the poka-yoke system* (Productivity Inc., Trans.). Cambridge, MA: Productivity Press. (Original work published 1985).

Schulmeyer, G. G., *Zero Defect Software* (New York: McGraw Hill Book Co., 1990).

Tribus, M. (1987, Spring). “The quality imperative,” *The Bent of Tau Beta Pi*, pp. 24-27.

“The Push for quality,” *Business Week* (1987, June 8). pp. 130-143.