Measurement-Driven Quality Improvement in the MVS/ESA Operating System

Jeffrey S. Poulin and David D. Brown

Loral Federal Systems–Owego and
International Business Machines Corporation

Abstract

Achieving gains in software quality requires both the use of software metrics and the desire to make measurement-driven process improvements. This paper describes the experiences, changes, and kinds of metrics used to drive quantifiable results in a very large and complex software system. Developers on the IBM Multiple Virtual Storage (MVS) operating system track, respond, and initiate better ways of controlling the software development process through the use of metric-based defect models, Orthagonal Defect Classification (ODC), and object-oriented techniques. Constant attention to progress with respect to the measurements and working toward goals based on the metrics show that measurement-driven process improvement can lead to noticeably better products.[1]

1.0 Introduction

The MVS/ESA Operating System (Multiple Virtual Storage / Enterprise Systems Architecture)[2] serves as IBM's premier operating system in the large system environment, addresses the commercial transaction processing, batch, and data processing needs of intermediate to high-end computing installations. It has a history of supporting robust, "mission-critical" commercial applications for many of the world's largest business and public institutions. MVS/ESA manages many application areas, including those involving very large databases and requiring on-line transaction processing, data sharing, resource utilization, and dependability.

In recent years MVS has added support for the Enterprise Systems Architecture with large data-in-memory techniques, as well as base support for highly-parallel computing, B1-level security, and Portable Operating System Interface (POSIX) compliant services. However, adding this function has caused MVS/ESA to nearly double in size in less than five years. With well over 2 million lines of code in the Base Control Program and more than ten thousand parts it represents one of the largest, most complex pieces of software in the world. In spite of the size and complexity of the environment, the MVS development team recently has achieved significant improvements in the overall quality of the product through measurement-driven techniques. This paper describes how measurements and quality goals play a key role in driving process and product improvement activities [7].

2.0 Background

As with many software development groups, the MVS development and test organizations have long used a variety of measurements to monitor software development. With the support of in-house tools, programmers routinely gathered and tracked process and software quality data during all phases of design, code, and test. In particular, results from design and code inspections included information such as person-hours spent in preparation, person-hours spent in the inspection, the amount of design/code reviewed, and the number of major defects found.

2.1 Data collection and tracking

Initially, the MVS team collected process and quality data in several independent databases driven by independent input tools. In recent years projects consolidated measurement and tracking information for products and product releases in an Integrated Project Support Environment (IPSE) software statistics database. This database contains all critical software management data on MVS and MVS sub-products, and allows us to effectively record and analyze metrics. Automated tools and triggers gather and automatically track information

[1] Proceedings of the *2nd International Symposium on Software Metrics*, London, UK, 24-26 October 1994.

[2] The MVS and MVS/ESA trademarks belong to the IBM Corporation.

across projects to determine a variety of information about the MVS/ESA product and the development process.

Not unlike the metrics databases in use by other companies [5], typical information includes the size and type of modules, which modules call which other modules, and change history information. In addition, we regularly monitor use and proliferation of parts from our common reuse library. For object-oriented software we track information such as number of classes and methods, class and method size, as well as usage ratios for the common class libraries.

2.2 Historical approach to quality measurement

Development organizations regularly used the results of measurements to assess the quality of individual projects, monitor the readiness of a release for shipment, and to make projections for the quality of the product. Typical measurements included release-to-release problem rates, problem severity distributions, unit test arrival and fix rates, number and size of design changes, and projected versus actual number of customer reported problems. Based upon accumulated historical data, a matrix showing inspections over time and a cumulative defect curve indicating both gross defects and defect arrival rates proved the most reliable metrics for projecting release quality.

2.3 Context for the quality initiative

During 1991 the MVS development team began a major quality initiative on the MVS/ESA Base Control Program. IBM announced the initiative in 1991 using the MVS/ESA SP3.1.3 product release (December 1989 availability) as a baseline. The first release delivered under the initiative, MVS/ESA SP4.3.0, became generally available in March 1993. The emphasis on quality resulted from several converging influences.

The first influence came from the marketplace; customers demanded even higher levels of availability for all their data processing needs. Rising expectations, driven by competitive pressures and the proliferation of around-the-clock, mission-critical applications left little leeway for even rare outages. Second, executives issued an companywide quality challenge which included assessments, inspections, and recognition based on quality improvement. The recognition included competing for the Malcolm Baldridge National Quality Award and achieving quality benchmarks based on the 10x/100x/6 Sigma model which had led to the success of quality programs at companies such as Motorola [6]. Third, development managers wanted to achieve certification of compliance with ISO9000 software production standards. ISO9000 certification indicates the organization can consistently produce software of the same quality, something customers recognize as an independent “stamp of approval.”

The following sections describe the quality initiative and how it brought about an increased emphasis on the software development process as reflected by the way we measure its many aspects. In the past we may have viewed a software defect too frequently as an personal error. While not removing personal accountability from designers, developers, and testers, a focus on process capability rather than individual culpability has added a dimension to defect assessment that we previously did not have Likewise, we have witnessed a significant trend toward the “democratization of software quality,” by which involvement in quality measurement and assessment has shifted from assurance groups and project managers to greater direct involvement by individual teams of programmers and testers.

3.0 Improving the Current Process

The quality initiative drove several practices and activities in a complimentary fashion. Each activity built upon and supplemented each other in such a way as to create a more complete picture of process and product defects. Of course, overall product quality entails many aspects of the product in addition to the absence of code defects. However, many of the measurable aspects of quality include defect reduction, and therefore the initiative included:

1. The use of defect models
2. Making more rigorous process definition and documentation
3. Increasing inspections and testing
4. Introducing Orthogonal Defect Classification (ODC)
5. Adding system test measurements

3.1 Use of defect models

The use of defect models that encompass the entire software development life cycle made an important step in moving toward a goal and measurement-driven approach to defect elimination. While we recognize that we must view quality as much more than merely an absence of defects, defect models allow us to view the entire process and to take a consistent view of gathering and tracking our data. It also provides a means for project teams to track data and, along with the project managers and Release Manager, set quality goals for each release.

As we consider defect injection and removal models, remember that we intend to not create the defect in the first place. The models only help us find where defects occur and lessen both their impact and the cost to fix them. In fact, we want to influence the shape of the defect curve by effecting changes which:

1. reduce the number of defects injected, or
2. move discovery of defects earlier in the development/test cycle.

Actions which prove effective in either of the above situations can result in improvements to the overall organization defect model and development process.

If we look at the progression through the development life-cycle, we gathered design and code inspection data, data on defects discovered during testing, and data from customer problems. Much of this information came from detailed results spanning years of experience. The act of creating a defect model caused us to go back and take a more consistent view of data so we could evaluate our progress towards quality goals. We wanted to know how many defects we had and how we should evaluate our progress.

Our basic defect models use historical data to form a statistical profile of the numbers of errors we expect to find at each phase of development and test. Each project prepares a defect profile, such as the one in Table 1, showing the expected number of defects by phase. The project then tracks the actual errors found throughout the life cycle. The composite numbers from all projects form a product level profile which we use to assess progress toward the overall quality objectives of the release.

Table 1. Sample Project Defect Model

	PD	**CD**	**MD**	**C**	**UT**	**FT**	**ST**	**GA**
Target	33	79	149	191	95	62	11	7
Actual	67	38	209	291	137	75	8	2

Note:

PD = Program Level Design
CD = Component Level Design
MD = Module Level Design
C = Code
UT = Unit Test, FT = Function Test, ST = System Test
GA = General Availability

Creating projected defect injection and removal models serves as a positive quality step in itself. The activity provides focus for the entire team on quality objectives and gives reality to the magnitude of the quality challenge. As such, it readily becomes the basis for goal setting and concrete improvement initiatives. However, the effort to create and track defect models did not readily appeal to programmers. Changing this attitude required ownership and pushing from designated process owners along with recognition from line management that the defect models represented their commitment to quality improvement activities.

The initial model analysis takes place at the kickoff meeting for the individual project. At this time the team sets goals regarding defects and related defect information. In addition, the release team also sets projections for the release based on the combination of defects removed from the previous release plus new defects injected by any new or changed code.

Figure 1 shows how the defect model projections for a 8,000-30,000 line pilot project [1]. In this case the programming team looked at data from similar projects within the operating system that ranged in size from 8,000 lines to 270,000 lines. After the team adjusted the data to allow for complexities, projected field defect rate, and other environmental factors, they plotted the defects per 1,000 lines of code *(defects/kloc)* on the vertical axis against the phases of the development life-cycle on the horizontal axis. *CLD, MLD, U/T, MCT, FCT, and A/T* denote the phases *Component Level Design, Module Level Design, Unit Test, Module Component Test, Function Component Test, and System Test,* respectively. IBM uses the acronym *APAR* (Authorized Program Analysis Report) to refer to a defect found in the field.

Using a 20 defects/kloc injection rate, the middle line on the graph indicates the total number of defects they

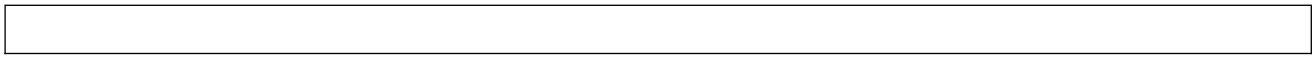

Figure 1. Initial Projections for Pilot Project

expect to identify at each phase. The upper and lower lines specify the statistically acceptable upper and lower limits for the expected number of defects. During development, programming teams regularly collect defect data and carefully track this information against the graph. Everyone on the project knows the team's progress. As programmers walk the halls of the laboratory, they can see the charts reflecting this progress posted on the walls where their development teams work.

For the MVS/ESA SP4.3.0 release, the Release Manager and Core Team regularly analyzed project defect models and actual defects during bi-weekly status meetings. They discovered several projects which tracked very favorably against their projections. However, they also singled out four projects where defect discovery significantly exceeded what they expected. The manager asked the project to initiate causal analysis on the reported problems and report back on analysis results and corrective actions.

3.2 Goal-driven inspections

During 1991, the MVS design, development, and test groups stepped up to a challenge of shipping the next release with 10 times fewer defects than the 1989 release (MVS/ESA SP3.1.3). As we started this *10x* project, we looked at the data we had historically collected and tried to get an assessment of how many latent defects might exist in the legacy code. This involved two separate assessments based on both service and development projections. We then broke out overall defect projections into "defect quotas" for individual component areas. We calculated the projected quality for all new code we planned to write, how many errors we would likely inject, and how many defects we would have to remove for us to meet our 10x objectives. The defect removal targets depended on factors such as the project size, complexity, and defect history. We then distributed the goals to the projects and began to routinely track progress against those goals at the component and product levels.

We implemented a variety of actions during this defect detection and elimination phase. Special expert reviews focused on known trouble spots and we built routines to parse for strings and look for situations typical of some types of errors. For example, in MVS the string "((" needs to precede a pointer. So, we wrote a parse routine that easily identified common errors associated with the "((," such as uninitialized pointers, unlabeled pointers, and even variables that do not act as pointers. By focusing on the these variables, the inspection teams also found other tangential errors, such as cases of sections of dead code caused by pointers to code in unrelated portions of the program. Where ever possible we applied these actions to newly developed code (code developed for the new release) so as to raise the overall quality of the new code as well as the existing base.

A lot of the 10x effort consisted of looking at how we could do the same job better. We held more inspections and walk-throughs, increased the intensity of our testing, and invested in quality software components as part of our reuse program. We soon achieved measurable results from this process activity. For example, through the use common macro libraries and reusable Building Blocks, test results showed about 9 times fewer defects during function test and 4.5 times fewer defects during component and system test (as measured by errors per line of code) in the reusable code [2]. We believed that we could achieve further significant results if we could capture more knowledge and information about our process and how we developed software.

As those activities continued, we held our first defect analysis sessions using Orthogonal Defect Classification (ODC) [3], [4]. ODC allowed us to say "besides the raw data, what else do we know about the problem?" For example, do we have quantitative data, or do we at least have qualitative data that points to process problems?

3.3 In-process measurements for defect classification

To add to our ability to identify, find, and correct errors, we started a pilot program using Orthogonal Defect Classification. The ODC method allows developers to establish and interpret a detailed model of their process, detect departures when they occur, and gives the necessary insights to make adjustments when needed. ODC uses defects to give feedback into the development process; because it uses in-process measurements we can use it at any stage of the life-cycle.

ODC works by having a team analyze each defect one at a time, capturing the defect type, the defect trigger, and the source, impact, and environment surrounding the defect.

Remember that the basic defect model tells the developers how many errors we expect to find at every phase of the life-cycle. However, if the actual results lie close to the target then we may not ask the right questions because we may think they did a good job based on having found the right *number* of errors. More often than not this does not tell the whole story. The basic defect model also has difficulty explaining why a developer might find an unexpectedly high or low number of defects.

If we want to know *why*, we have to ask if we did a good job and that means looking at what kinds of errors we found. If we found too many, we have to look at what kinds of errors we did find. If we found fewer errors than we expected, we have to ask if we feel comfortable with the quality of the inspections and reviews. In other words, the basic defect model provides an initial, limited set of information that can indicate important trends, such as if we consistently obtain more or fewer errors than we expect at each phase.

To explain the basic defect model we need to consider *context.* To do this we use ODC; it adds the third dimension. ODC helps us determine if we have things we should not feel comfortable with and if we have problems, whether the kinds of problems reflect important or relatively minor things. By classifying the errors we know not only how many errors to expect but also the *types* of errors and where they might occur. Most importantly, we can identify places where we might have missed finding certain classes of errors. Missing errors reflects the worst situation for our developers because it points to a process problem; in short, why did not we find the errors the first time?

To implement ODC, we look at defect types at every major phase, each of which may last a couple of months. For example, we examine and classify defects when we go from High Level Design (HLD) to Component Level Design (CLD), and from CLD to Code. Once we have enough data for statistical significance we use ODC analysis to reassess the defect model and to track our progress with respect to the model. Because we have data quantifying the types of errors, where they normally occur, and where we normally find them (e.g., inspections, walk-throughs) we know where to focus in terms of process changes and improvements.

Table 2. Mapping Defects to Process Improvements

Trend	**Process Problem**	**Process Change**
Too many missing defects; too few incorrect defects	Lack of communication between sub-teams	Repeat inspections; conduct teach-the-team sessions
Too many capability defects; too few performance and usability defects	Nonfunctional areas of design not adequately addressed	Emphasize non-functional areas in succeeding phases
Function; lack of traceability to component design	CLD done during MLD on discovery of missing requirement	Reinspect design documents
Function; missing associations	Incomplete recovery of design	Emphasize design recovery during inspections
Backward compatibility defects	CLD does not address backward compatibility	Complete CLD and reinspect
Function association defects	Vague requirements cause iteration between MLD and CLD	Conduct functional requirements inspection

Table 2 gives scenarios that map undesirable trends, the likely process deficiency that caused the trend, and a corrective action to minimize the deficiency [1]. For example, we might analyze the results of our second design review and determine that the *types* of errors we see do not fit the defect model profile. As a result, we might return to the design review and conduct additional inspections to determine the reason for the discrepancy. The ODC measurements help us make that determination and help identify what actions to take to correct a problem.

3.4 System test measurements

In addition to standard test measurements used in the past, the System Test organization developed new measures to gauge System Test entry criteria and readiness for product ship. The new metric, called a *Quality Stability Index (QSI),* provides a quantitative method to assess the readiness of a product to enter system test, to track a product's stability throughout the life cycle, and to serve as an early indicator of expected product quality when it reaches the field. The QSI combines a number of software stability factors including defect removal activity, test coverage, defect discovery rates, and error backlog into a weighted composite value that ranges from 1 to 100.

The QSI establishes a quality rating for a system based on factors in addition to traditional methods which often depend solely on test results. As shown in Table 3, test organizations set minimum thresholds for the QSI; for example, entry to System Test requires a system $QSI = 50$ and exit requires a $QSI = 90$. From a release standpoint, the QSI provides an additional quality tracking tool and an additional means of confirming overall release quality. If the product does not meet the thresholds established for the QSI metric, additional process activities must take place to bring the QSI within the desired limits.

Table 3. Tracking the QSI index through System Test

Test Cycle	Projected Percent Complete	Actual Percent Complete	Projected QSI Rating	Actual QSI Rating
1	0	0	50	45
2	8	4	54	51
3	20	15	60	55
4	50	45	69	66
5	80	75	78	74
6	100	100	90	92

4.0 Making Fundamental Process Changes

The strategy to achieve additional quality improvement for MVS included introducing object-oriented concepts to the operating system. We got to 10x improvement by doing a lot of process discovery activities, such as identifying and fixing errors in the product base. These discovery activities included reviewing problem code and intensifying the old way of software development. To improve on 10x we knew we needed to go beyond the inherent complexity issues surrounding how we develop our code; this meant using new techniques that fundamentally change the software process.

4.1 Introducing object-orientation

We decided that object-oriented design and the construction of high quality reusable frameworks would play a critical role in the quality strategy. To support the introduction of objects into the legacy system, we developed an environment called CASE/390. CASE/390 provides the software and tools to integrate object technology into a large, complex product with millions of lines of procedural code. The tools allow programmers to develop using object-oriented language features and generate code that will run with the existing procedural base.

Among the many advantages of object orientation, the CASE/390 environment helped us by the amount of additional preprocessing conducted to create and encapsulate objects. Because of the intense performance requirements in an operating system kernel, coding sometimes takes place in languages that do not support common software engineering features. For example, CASE/390 introduced strong typing, which alone helped locate several different classes of errors.

Currently, about 30,000 lines of fielded software comes from CASE/390 and about 225,000 lines will appear in the next MVS release. The following case studies give experiences from adding object orientation to a large system:

Case Study I. This project involved approximately 6,000 lines of generated procedural code and about 3,000 lines of code written by a team of 2-3 programmers. The project experienced 5 errors in in the CASE/390 code in the field, all 5 caused by division by zero and each inserted by a programmer's misunderstanding of the allowable range of values that a customer could specify (i.e., the externals of the system allowed 0 as a input value but the code did not handle that case). Comparing the internal error rates between the CASE/390 and non-CASE pieces, internal error rates for the CASE/390 code came in at 50% of the error rates for the procedural portion of the code. Historical error rates for the component ranged anywhere between 2 and 6 times the rates we measured for the CASE/390 code. Although about 40% of the historical errors come from addressability problems, no addressability errors appeared at any stage of this project. The developers went through about 16 iterations of the code; each iteration lasted an average of 3 days.

Case Study II. This project involved approximately 20,000 lines of procedural software and a team of 3-5 developers. This project also consisted of a mixture of CASE/390 and manually written procedural code. The CASE/390 software experienced approximately half the errors of the procedural software. In fact, this project has had no errors in the object software reported from the field.

In general, teams using objects and the CASE/390 environment experienced a 2.9 times productivity increase and a 40% decrease in error density in applications and extensions to MVS they developed using the object-oriented approach. As more and more programmers reused the CASE/390 objects, they identified and fixed errors, or even built intelligent rule-based mechanisms into the preprocessor to check for the types of situations with which programmers routinely have difficulty. This means testing goes faster because developers have less rework and objects do not need retesting once they have proven to work. Finally, although we add new classes for each project, most of the existing function comes from a relatively small (approximately 275 classes) class library that provides a core set of services.

The key lies in deciding on which metrics to focus. Technology transfer requires constant attention and feedback; we use our metrics to track quality improvement, of which CASE/390 serves as one agent to that goal. Although we have not quite reached the point where we can track class defect data in CASE/390, we track the same metrics for the CASE code as we do for traditional development and use that data in our quality program.

4.2 The metric-driven process

The combination of object-oriented techniques, defect analysis, and measurement-based process improvement has shown tremendous results. Furthermore, the feedback has helped with our site-wide insertion of the technology because as other teams see the results they want to learn more and ultimately try it. For example, one project used a defect model and specific process feedback actions to include ODC analysis. A follow-on team saw the measured results of this project and readily adopted the process improvements.

As a result of the various measurement activity on MVS/ESA SP4.3.0, the Release Manager commented that available data exceeded that of any prior experience the manager had with a development project. The data directly related to release progress and provided the ability to have a daily view in terms of inspections and defect reports. The manager could see changes, trends, and the results of actions taken to improve quality and meet schedule commitments. The core team for the release also benefited from the tracking tools and were able to detect problem areas from the data. As a result, they became adept at isolating problems relating to:

- schedules and commitments
- design and code quality
- process effectiveness
- problem receipts, resolutions, and backlog

The end result of a metric analysis identifies a set of actions. These actions include going back and fixing a problem, or adopting a set of things to change for the next iteration so that we do not make the same mistakes again. This defect prevention step often consists of committing to looking at something more carefully, paying more attention to a particular trouble area, or establishing inspection procedures to identify certain potentially volatile kinds of situations. The metrics also allow us to focus on a specific section of the code that may have revealed itself as possibly having an unusually high percentage of problems.

If we have to go back and analyze a section of code, the metrics allow us to make the rework a much more focused activity. For example, if during the code review we identified a series of problems dealing with serialization errors, we will go back and hold a serialization review as opposed to a general review. This allows us to focus on serialization; we might hold reviews for other specific areas such as user interface errors if the metrics point to them as potential problems.

Another way the numbers and defect analysis can help us comes in a situation like when the new code has a lot of backward compatibility function and we did not identify many compatibility type errors during our inspections and test. We may have done an exceptional job during design or we may have a problem because the team did not look at that type of error well enough. Again, the measurement results allow us to conduct a very focused review to determine why we have deviated from our model and determine if we require a process change to fix it.

We now have and maintain defect histories for every product and know the expected number of defects, by phase, throughout the life-cycle. These raw numbers allow us to construct a defect model by which we can determine and track our progress on a project. We standardized these defect models for MVS Release 4.3 and started using them on CASE/390 projects. The benefits come from the emphasis on tracking these metrics and taking actions to correct suspected problems identified by the metrics.

5.0 Conclusion

The emphasis on measurement-based process improvement resulted from the need to quantify our progress in meeting quality goals and in the desire to show the MVS programming team the results of our quality effort. Although we historically collected all kinds of data related to our process, in the past our development teams did not look at and analyze the data. We routinely provided basic reports on numbers of errors and the severities of those errors, but it took an emphasis on metrics to give us a reason to use the data in defect models and track the metrics to a goal. The defect models have helped focus developers on ferreting out defects early in the process, improving overall quality, and have helped us maintain that quality focus. Measurements-based defect models and the use of ODC form the core of the 10x quality program and have had across-the-board affects in terms of quality and product improvement.

The long-range plan for greater improvement depends on object technology and constant attention to results based on quantifiable data. Initial reports from CASE/390 show significantly higher quality in object-oriented code as compared to procedural code developed by the same teams. In some cases the quality improvement reflects a factor of more than 2.

The first MVS release under the quality initiative, MVS/ESA SP4.3.0, became available in March 1993 and has received high marks for improved quality, ease of installation, and improved serviceability. In addition, the Mid-Hudson Valley Programming Laboratory earned the ISO9000 process certification during its evaluation in the Spring of 1993. These kinds of benefits, made possible through the use of metric-based defect models and an emphasis on progress with respect to the models, show that metric-based process improvement can help lead an effort for improved quality and better products.

6.0 Acknowledgements

We would like to thank Dr. Jarir Chaar of the IBM Thomas J. Watson Research Center, and Bev Wick and Eric Tarver of the Mid-Hudson Valley Programming Laboratory for their assistance during the preparation of this paper.

Author's address: Jeffrey S. Poulin, Ph.D., poulinj@lfs.loral.com; MD 0124, Loral Federal Systems–Owego, New York, 13827. David D. Brown, Ph.D., dbrown@pokvmcr3.vnet.ibm.com; International Business Machines Corporation, MD P413, Poughkeepsie, New York, 12602.

7.0 References

[1] Bhandari, Inderpal, Michael Halliday, Eric Tarver, David Brown, Jarir Chaar, and Ram Chillarege, "A Case Study of Software Process Improvement During Development," *IEEE Transactions on Software Engineering,* Vol. 19, No. 12, December 1993, pp. 1157-1170.

[2] Bauer, Dorothea, "A Reusable Parts Center," *IBM Systems Journal,* Vol. 32, No. 4, 1993, pp. 620-624.

[3] Chaar, Jarir, Michael Halliday, Inderpal Bhandari, and Ram Chillarege, "In-Process Metrics for Software Inspection and Test Evaluations," *IEEE Transactions on Software Engineering,* Vol.19, No.11., November 1993, pp. 1055-1070.

[4] Chillarege, Ram, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong, "Orthogonal Defect Classification- A Concept for In-Process Measurements," *IEEE Transactions on Software Engineering,* Vol.18, No.11., November 1992, pp. 943-956.

[5] Daskalantonakis, M.K., "A Practical View of Software Measurement and Implementation Experiences within Motorola," *IEEE Transactions on Software Engineering,* Vol. 18, No. 11, November 1992, pp. 998-1010.

[6] Mitchell, B., "The Six Sigma appeal (SPC)," *Eng. Manage. J. (UK),* Vol. 2, No. 1, Feb. 1992, pp. 41-7.

[7] Kan, S. H., et al., "AS/400 Software Quality Management," *IBM Systems Journal,* Vol. 33, No. 1, 1994, pp. 62-88.