COMPUTER 0018-9162/96/$5.00 © 1996 IEEE
Vol. 29, No. 11: NOVEMBER 1996, pp. 78-87

Contact Thomas Drake at 900 Elkridge Landing Rd., Ste. 100, Linthicum, Md. 21090; drake@bah.com.

Measuring Software Quality: A Case Study

Thomas Drake
National Security Agency

To ensure cost-effective delivery of high-quality software, NSA has analyzed effective quality measures applied to a sample code base of 25 million lines. This case study dramatically illustrates the benefits of code-level measurement activities.

The National Security Agency's mission is to provide support for the security of the United States. Over the years, the Agency has become extremely dependent on the software that makes up its information technology infrastructure. NSA has come to view software as a critical resource upon which much of the world's security, prosperity, and economic competitiveness increasingly rests. If anything, dependence on software and its corresponding effect on national security makes it imperative for NSA to accept and maintain only the highest quality software. Cost overruns, or software systems that are defective or of low quality, can impose a significant burden on national security and NSA's mission.

The NSA spends many hundreds of millions of dollars annually on software development and maintenance. An internal NSA study shows that an average software project generates only seven to eight lines of delivered code per person per day, at a cost of approximately $70 per line (assuming a $140,000 annual loaded cost for each software developer). Viewed in these terms, it becomes clear that we need to lower software costs by finding ways to improve productivity while still maintaining quality.

NSA is no more immune than the rest of the software industry to the problems of low-quality software. I myself spent the first five years of my contractual relationship with NSA on software projects that were consolidated, cut back, or simply stopped at the government's discretion. Most of the problems on those projects resulted from highly unstable, error-prone software. Why does this happen?

It happens because software development is intensely manual in nature. A lot of time and effort goes into coding, and software inevitably falls victim to rushed schedules, constantly changing requirements, poor process, failure to adhere to software engineering practices, and indifferent coding practices and standards. Yet software quality requires a defined process supported by automated technology and the application of a base set of measures.

The rub is the difficulty in measuring software—especially the specific task-level activities of software development executed by a programmer/engineer. The traditional algorithms and metrics from the hard engineering disciplines either come up short when applied to software or are easily misapplied. Software engineering is a discipline that is largely nonparametric and resistant to traditional modes of analysis. Unquestionably, we need common methods of determining the business value of software. We must also generate and promote the human factors and activity statistics necessary to manage and steer software development.

So what is NSA doing about software quality?

SOFTWARE ENGINEERING APPLIED TECHNOLOGY CENTER

The NSA organization responsible for addressing these needs is the Software Engineering Applied Technology Center. The SEATC's goal is to reduce maintenance costs while improving software development. In addition, NSA has formed partnerships with other government agencies and industry consortiums, the Software Engineering Institute, and vendors who supply automated tools and support technology for software development. Its aim is to discover the best current practices in government and industry and bring knowledge of them to NSA organizations.

A key initiative addresses software quality engineering. The benefits of structured testing and quality assurance activities throughout the entire software life cycle are often overlooked, but they *are* a key part of a complete software engineering process.

Testing activities provide key software metrics and feedback mechanisms for defining and improving software products across a broad spectrum of software development activities at NSA. Process-level activities involve technical test management support and software quality assurance program implementation.

After nearly three years of measurement and quality assurance activities, the SEATC has collected and analyzed the results on some 25 million lines of C, C++, Fortran, and Ada code. We have developed a streamlined set of code-level release criteria that we apply to code written at NSA organizations. This 25 million lines of code is the sample drawn from a population of more than one billion lines of code at NSA. The result is a highly correlated set of measures that promote high-quality processes where they matter most—at the code level.

The SEATC software quality measures

We use two primary measurement activities to derive our code-level release criteria. The first is code metrics analysis: We measure development productivity indicators, predictability measures, maintainability indicators, essential quality attributes of the code, and "hot spots," and we identify overly complex modules that need additional work. Specific code metrics include Halstead's software science measures (purity ratio, volume, and effort), McCabe's cyclomatic complexity, and functional density analysis.

The second measurement activity is coverage analysis, which centers on "inside-the-code" analysis (or decision-level metrics), testability indicators through executable path analysis, and predictive performance analysis based on the number of segments per path.

These measures constitute what we call critical or essential measures.[1] (See the "NSA SEATC critical measures" sidebar.) We have found them valid for measuring most high-order code produced in support of NSA and for determining code characteristics, such as:

- likelihood and frequency of errors,
- overly complex components that might require reengineering,
- allocation of resources for testing,
- difficulty of maintenance,
- predictive performance indicators, and
- optimization level.

The primary SEATC technique for determining software quality is to analyze the changes (deltas) to the code. We track this measure against changes in other measures over time via repeated milestone and turnover "snapshots." These snapshots provide a feedback loop that lets us improve not only the quality of the software, but that of the development process as well.

The SEATC's code quality index

On the basis of the normalized use of simple standard deviation and frequency distribution curves, the SEATC has developed a code quality index to measure the relative stability, robustness, size, and predictive performance of a typical function in a high-order language. We used an integrated set of static quality indicators and correlated them to the results of dynamic testing of the same function running as a compiled executable in the real-world environment. The mean for each metric represents the "ideal" normalized value, and the quality index encompasses approximately three standard deviations on each side of the mean. Correlations between static and dynamic testing run between 85 and 95 percent. Acceptable ranges for each metric are detailed in Table 1, along with adjustments for graphical user interface code generators and for embedded database procedure calls. (The ranges in the table encompass approximately only one to one and a half sigmas.) In addition, these range values must be baselined for a particular application domain.

Table 1. The SEATC critical measures and benchmarks.

Measure	Formula	Standard	Ideal	Range (min-max)	Comments
Purity Ratio	$PR = N\char`^ / N$	1.0	>1.25	0.85-1.25+	Automated GUI and DB procedure call range: 0.5-1.0+
Volume	$V = N \times \log_2 n$	3,200	<1,000	<3,200-4,500	Automated GUI range: <3,200-10,000 DB procedure-call range: <3,200-7,500

Effort	$E = V / L^\wedge$ $L^\wedge = 2 / n_1 \times n_2 / N_2$	300,000	100,000 or less	<100,000-450,000	<500,000-5,000,000 for GUI code <300,000-1,000,000 for DB procedure calls
Cyclomatic complexity (VG1)	$V(g) = e - n + 2$ (e = edges and n = nodes)	10	<10 (6-7 for C++)	2-15	2-10 for GUI code <10-25 for DB procedure call code Extended cyclomatic complexity (VG2) adds ANDs and ORs + 1 to the number of decisions. Maximum is 15 per function, or VG1 × .25
Functional density	LOC / FP	62	36	<25-150	A function point (FP) is defined here as a function in a high-order language; at a minimum, a function is based on an input (entry point) thatresults in an output (exit point).
Executable path analysis per function	Total one-trip path count per function	<100	20-50	50-1,000	1,000 is an outside maximum for normal code. Range for GUI tool code: <1,000-50,000 Range for DB procedure call code: <1,000-20,000
Average number of logical branch links per path		<50	15-25	<15-50	We've found a very high correlation between high average branch links per path and performance degradation.
Estimated time to develop function in in hours	$T^\wedge = E / S / 60 / 60$ (Conversion to hours entails dividing by 60 sec. per min. by 60 min. per hr.)	4.5	1.5-2.5	1.0-7.0	*S*= Stroud index (sliding scale) The normal index is 18 for "average" programmers. The index number for beginning programmers is 5-10 and up to 40+ for highly efficient programmers.
Predicted errors / KLOC	$B^\wedge = V / EO/(\text{LOC}/1{,}000)$ *EO* = 3,200 (average number of mental comparisons made before a mistake is generated)	4	<3	<3-8	Low-quality code has 15-40 errors per KLOC and up to 100+ per KLOC for "pathological" code.
Size	LOC per function	62			
Percent of comment lines per function	Comments / LOC - Blank lines	60	60+	40-60+	
Executable statements per function	Semicolon count for C and C++	<50	15-30	<10-50	
Span of reference for variable		10-12			Average maximum number of lines between references for each variable's assignment and use

Legend—n_1 = number of unique operators; n_2 = number of unique operands; n = number of unique operators and operands (represents the essential vocabulary of the language); N_1 = total number of operators; N_2 = total number of operands; N = length as determined by $N = N_1 + N_2$; $N^\wedge$ = predicted length as determined by $[n_1 \times \log_2(n_1)] + [n_2 \times \log_2(n_2)]$.

C CASE STUDY

From this 25 million lines of code, I have drawn up a case study to dramatically illustrate the benefits of applied quality assurance and code-level measurement activities.

About three and a half years ago, an NSA organization needed real-time network support. This software was designed to analyze the parameters of certain kinds of data streams and, on the basis of that analysis, send the data to a workstation on the same network. The software was to be developed, integrated, and delivered in six months.

During these six months there were no code inspections, walkthroughs, separate testing activities, or process controls. When the code was installed for operational testing, it failed abysmally—not because it did not work, but because the critical function, the delivery of data, took four or five hours. The users deemed this unacceptable, even though technically the software worked.

The support organization assigned its senior developer to fix the code. She spent some two weeks trying to fix it before determining that it would have to be reengineered. Management (the root cause of much that afflicts software acquisition) was getting very antsy. The project was now way behind schedule. They insisted on keeping the old software on the system even though it caused analysts to wait hours for data.

This went on for some weeks until the senior developer simply decided to reengineer the software anyway. After six weeks of effort, the new code was reintegrated. Data delivery that took four or five hours now took only seconds. The users were much happier, but management was still very upset because it took so much more effort to deliver the same functionality.

The developers had no mechanism for demonstrating quantitatively what they already knew anecdotally about clear and obvious differences in performance between the two functions. The senior developer in charge of the reengineering effort asked the SEATC to use its tools to analyze the code. Within two hours we were able to demonstrate very clearly why the new code was vastly superior in quality to the old code.

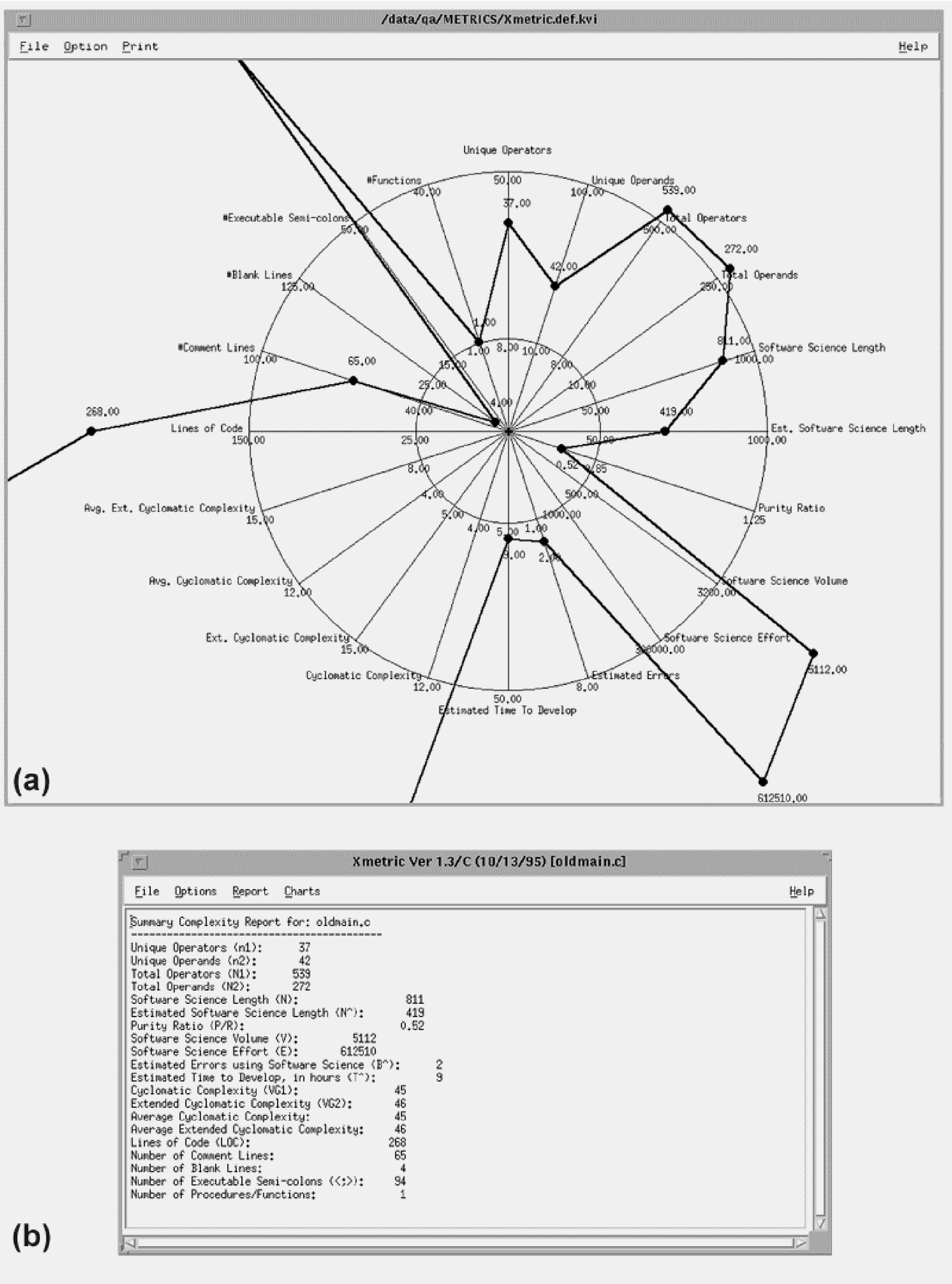
We made this case by combining static metrics analysis with static coverage branch-level analysis at the individual function level, using two commercial, off-the-shelf tools.

For the metrics analysis, we used the UX-Metric C language version 2.2 analyzer engine from SET Laboratories, supplied with the version 1.3 Metric tool from the Software Research Software TestWorks Advisor tool suite, which added Kiviat diagram graphing analysis. The metric engine assumed that the code was ANSI C compliant and/or Kernighan and Ritchie with the approved extensions.

For the coverage analysis, we used the TCAT (Test Coverage Analysis Tool) version 8.2 code analyzer from the Software TestWorks Coverage tool suite, using version 2.7 of the Xdigraph directed graph (digraph) display technology. Both tools ran on a Sun Sparc platform under SunOS 4.1.3.

Static metrics analysis

First, we analyzed the oldmain.c file and generated the complexity report represented by the Kiviat diagram in Figure 1a. A Kiviat diagram is represented by two circles. The inner circle is the minimum range value for each variable measure; the outer circle is the maximum range value. Each spoke of the wheel is assigned a measure. You simply connect the dots and analyze the pattern. The pattern should be a circle generally fitting between the inner and outer circles. A metric that falls below or above the circles flags a possibly suspect or at-risk portion of code.



```
Summary Complexity Report for: oldmain.c
------------------------------------------
Unique Operators (n1):       37
Unique Operands (n2):        42
Total Operators (N1):       539
Total Operands (N2):        272
Software Science Length (N):                811
Estimated Software Science Length (N^):     419
Purity Ratio (P/R):                        0.52
Software Science Volume (V):          5112
Software Science Effort (E):        612510
Estimated Errors using Software Science (B^):        2
Estimated Time to Develop, in hours (T^):            9
Cyclomatic Complexity (VG1):               45
Extended Cyclomatic Complexity (VG2):      46
Average Cyclomatic Complexity:             45
Average Extended Cyclomatic Complexity:    46
Lines of Code (LOC):                      268
Number of Comment Lines:                   65
Number of Blank Lines:                      4
Number of Executable Semi-colons (<;>):    94
Number of Procedures/Functions:             1
```

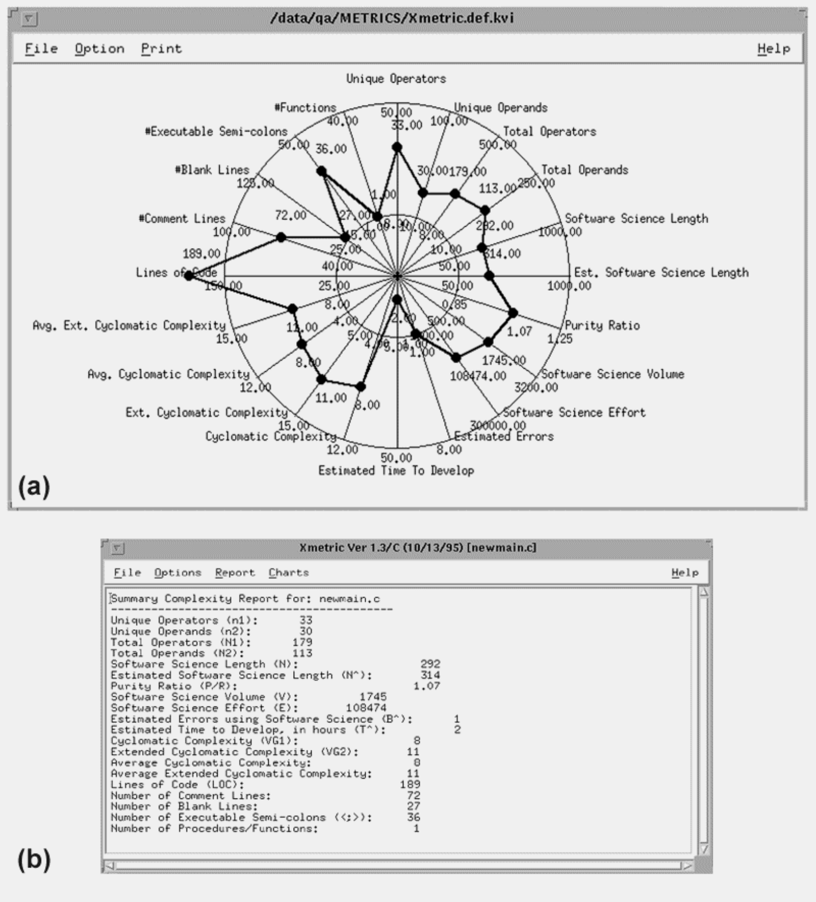
Figure 1. Complexity report for oldmain.c file represented by (a) a Kiviat diagram and (b) a summary report. The pattern depicted in Figure 1a is called the "flying Albatross syndrome" and suggests extremely volatile code.

In tracking the relationships of the various measures over some 25 million lines of code, we have discovered some patterns. We call the pattern in Figure 1a the "flying Albatross syndrome." Its tail is the number of executable lines or "live code." The oldmain.c main function has about three times more lines than it should. The right wing is the complexity measure. These measures are off the scale. The head is the volume and effort numbers. And the left wing is the number of operators and operands, which in this case is very large, translating into a very low purity ratio. Naturally, the bird is flying south in a crash dive position. This particular pattern has emerged time and time again in code that is at substantial risk.

The SEATC has calculated convenient sigma ranges for each of the critical measures. The first range encompasses about one standard deviation to a standard deviation and a half. In this example, the purity ratio is 0.52 as opposed to the minimum standard ratio of 0.85. This indicates that the code is highly unoptimized—that is, it has much more code than necessary to implement the functionality. This is further corroborated by a volume of 5,112, indicating that the program is much larger than normal (3,200). Effort is more than 600,000, or twice the maximum effort "permitted." Finally, complexity is 45, four-and-one-half times the accepted norm for most high-order code. In short, this code is probably unstable and/or volatile and was developed over an extended period of time. The SEATC will report any function as risky if it falls outside the accepted range values for any one of these four critical measures.

Figure 1b shows the summary report generated against the oldmain.c main function. The estimated time to develop this function alone is nine hours—six times the norm. And the number of comment lines is only 65—a percentage of less than 24 percent instead of the standard 60 percent.

The SEATC then ran the same measures with the same parameters against the newmain.c main function. Figure 2a shows a very different Kiviat diagram: With the exception of lines of code, all other measures fall within bounds. The purity ratio is 1.07, or better than ideal. Volume is 1,745 or about 54 percent of maximum. Effort is about 108,000 or about one-sixth the value for the oldmain.c and a third of the maximum, and complexity is eight—actually better than the standard.



```
Summary Complexity Report for: newmain.c
------------------------------------------
Unique Operators (n1):       33
Unique Operands (n2):        30
Total Operators (N1):       179
Total Operands (N2):        113
Software Science Length (N):                   292
Estimated Software Science Length (N^):        314
Purity Ratio (P/R):                           1.07
Software Science Volume (V):          1745
Software Science Effort (E):        108474
Estimated Errors using Software Science (B^):      1
Estimated Time to Develop, in hours (T^):          2
Cyclomatic Complexity (VG1):                  8
Extended Cyclomatic Complexity (VG2):        11
Average Cyclomatic Complexity:                8
Average Extended Cyclomatic Complexity:      11
Lines of Code (LOC):                        189
Number of Comment Lines:                     72
Number of Blank Lines:                       27
Number of Executable Semi-colons (<;>):      36
Number of Procedures/Functions:               1
```

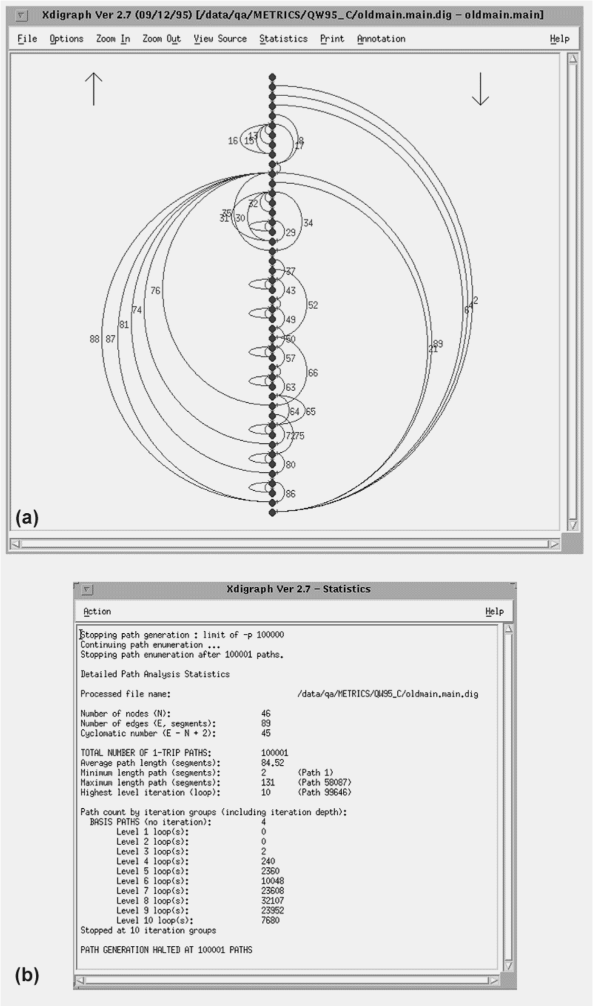
Figure 2. Complexity report for the newmain.c file represented by (a) a Kiviat diagram and (b) a summary report. Here, the measures fall within bounds, indicating stable code.

The estimated time to develop was only two hours, or about 4.5 times less than the oldmain.c file. It is also important to note that the ratio of the actual time to develop (six months) and the reengineering time to develop (one-and-a-half months) is almost the same as the estimated development time reported by the metric tool. Let us examine Figure 2b. The summary report reflects only 189 lines of code, 72 of which are comments, for a comment percentage of almost 50 percent—twice as high as the old code from the oldmain.c file.

Coverage analysis

We must point out, however, that metrics are not the only way to determine code quality. The SEATC also ran coverage analysis against the old and new code. It was this coverage analysis that predicted the performance, maintainability, and testability of both main functions.

Figure 3a shows a directed graph (control flow graph) of the old main function. The entry point is the top node, and the exit point is the bottom node. Each node represents a statement in the code and is linked to other statements by segments, as represented by the linking arcs. The graph is read top to bottom, right to left. Note that nodes 2-4 are hardcoded and result in immediate exits. Note also the large number of grouped statements linked to another group of statements only by an isolated link. This represents an "island effect" and strongly indicates a lack of design: Get something to work, code some more and get it to work, and so on. The pattern generated by the digraph reflects a very complex, low-performance module.



```
Stopping path generation : limit of -p 100000
Continuing path enumeration ...
Stopping path enumeration after 100001 paths.

Detailed Path Analysis Statistics

Processed file name:                            /data/qa/METRICS/QW95_C/oldmain.main.dig

Number of nodes (N):                  46
Number of edges (E, segments):        89
Cyclomatic number (E - N + 2):        45

TOTAL NUMBER OF 1-TRIP PATHS:         100001
Average path length (segments):       84.52
Minimum length path (segments):       2        (Path 1)
Maximum length path (segments):       131      (Path 58087)
Highest level iteration (loop):       10       (Path 99646)

Path count by iteration groups (including iteration depth):
  BASIS PATHS (no iteration):         4
        Level 1 loop(s):              0
        Level 2 loop(s):              0
        Level 3 loop(s):              2
        Level 4 loop(s):              240
        Level 5 loop(s):              2360
        Level 6 loop(s):              10048
        Level 7 loop(s):              23608
        Level 8 loop(s):              32107
        Level 9 loop(s):              23952
        Level 10 loop(s):             7680
Stopped at 10 iteration groups

PATH GENERATION HALTED AT 100001 PATHS
```

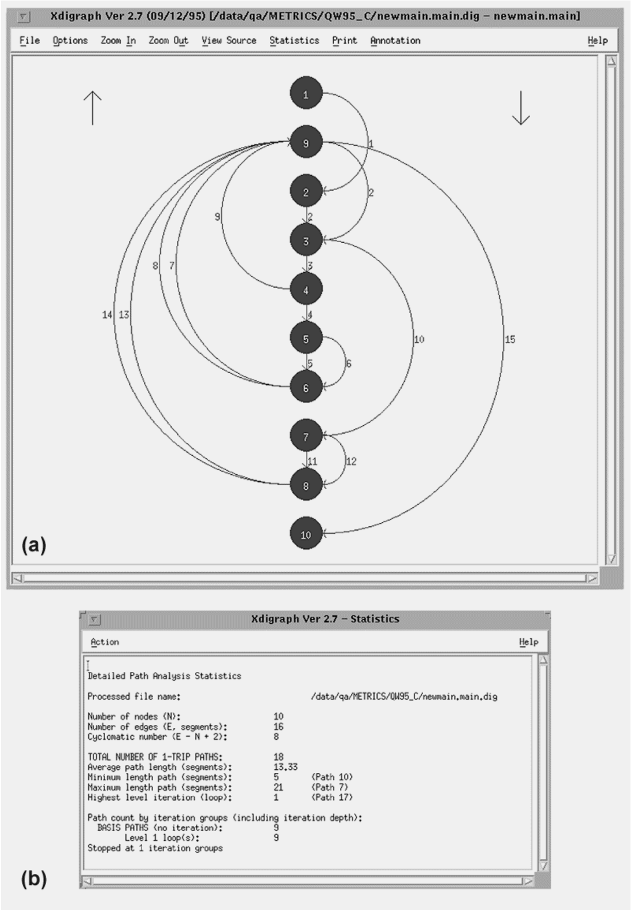
Figure 3. (a) A directed graph (control flow graph) of the old main function and (b) summary report of the statistics generated against the digraph. Each node represents a statement in the code and is linked to other statements by segments, as represented by the linking arcs. The pattern generated by the digraph reflects a very complex, low-performance module.

Figure 3b shows the statistics generated against the digraph. There are at least 100,000 one-trip executable paths. And this is by no means the total number. The original analysis on oldmain.c was executed on a Sparcstation 2 running 32 Mbytes of RAM and 32 Mbytes of swap space. After half an hour it ran out of memory after reporting at least 160,000 one-trip executable paths. When we put it on a Sparcstation 20 with 64 Mbytes of RAM and allocated 2.1 Gbytes of swap space from the network, it ran out of memory after almost 2.5 hours and reported more than eight million one-trip executable paths.

The normal maximum number of executable paths per function is 100. The extreme outside maximum is 1,000. The ideal number of paths is 20-50 on average per function. From a testability perspective, this particular function with its more than eight million paths cannot be tested. From a maintenance perspective, the program is essentially unmaintainable.

Predicting a module's performance is also important. Figure 3b shows that average path length based on the number of segments/links is more than 84, whereas the outside maximum should be 50. A lot of logical links per path are used to process data coming into this function, explaining why this function took upwards of five hours to complete.

The SEATC's coverage analysis of the newmain.c file had dramatically different results. Although the functionality shown in Figure 4a is exactly the same as that in Figure 3a, there are fewer node points and the function's structural appearance is much simpler. Most important, the number of executable paths is only 18, not the eight million plus for the old main. In addition, Figure 4b shows that the average path length for the new module is only 13 versus 84 for the old module. It was much easier to test newmain.c given the far smaller number of paths to cover.



```
Detailed Path Analysis Statistics

Processed file name:                        /data/qa/METRICS/QW95_C/newmain.main.dig

Number of nodes (N):                  10
Number of edges (E, segments):        16
Cyclomatic number (E - N + 2):        8

TOTAL NUMBER OF 1-TRIP PATHS:         18
Average path length (segments):       13.33
Minimum length path (segments):       5       (Path 10)
Maximum length path (segments):       21      (Path 7)
Highest level iteration (loop):       1       (Path 17)

Path count by iteration groups (including iteration depth):
  BASIS PATHS (no iteration):         9
        Level 1 loop(s):              9
Stopped at 1 iteration groups
```

Figure 4. (a) A directed graph (control flow graph) of the new main function and (b) summary report of the statistics generated against the digraph. The digraph has fewer node points and exhibits a much simpler structural appearance.

Armed with these results, the senior developer was able to return to management and quantitatively demonstrate the stark difference in quality between the old and new functions. In the intervening years, we have made such code analysis an integral part of the SEATC's software development process.

THE PATHOLOGY OF SOFTWARE DEVELOPMENT

Pareto's Law states that 20 percent of the code will contain 80 percent of the problems. Based on the 25 million lines of code formally analyzed to date, we have found that 10-15 percent of the modules will have 70-80 percent of the problems. For NSA, Pareto's law is closer to some 13 percent of the code accounting for close to 90 percent of the problems, with some 2.5 percent of the total code accounting for 95 percent of the most critical showstopper and functional disconnect errors. This path- ological code *must* be identified for risk analysis.

However, none of these measures are silver bullets. It is the combination of these measures applied over time and against a statistically significant sample size that becomes useful. In this respect, metrics are like the actuarial tables used by insurance companies to assess life insurance premiums. They provide the ability to manage risk and determine the code's essential quality level.

SOFTWARE RISK MANAGEMENT

As a result of this kind of analysis, the SEATC is now promoting a series of measures related to defect rates, code/system scrap rates, workflow and process measures, and cost measures related to cost-per-unit of code based on level-of-effort calculations, maintenance resource analysis, and productivity measures.

The whole intent of NSA's software quality program is to reduce risk by managing it through the analysis of measurement data applied to both process and development tasks down to the functional code. NSA is investing in the long-term software process and programming environments to ensure the long-term stability and operational use of its software base. The national security stakes are just too great.

The NSA SEATC created a comprehensive software process improvement effort through a series of strategic partnerships and pilot programs that allow the establishment, measurement, formalization, and improvement of the full life cycle software process.

The SEATC uses the SEI Capability Maturity Model as the basis for this initiative. In addition, we use the Experience Factory developed by NASA's Software Engineering Laboratory as the mechanism for evaluating software improvement initiatives. The Experience Factory model allows NSA to gather objective measurements for software development organizations and analyze them to determine the effectiveness of development techniques and document their costs and associated level of effort. Reengineering, reuse, integrated software development environments, object-oriented technology, and training are additional SEATC initiatives to promote and sustain high-quality software.

CONCLUSION

Many people talk about market share and software content, but not in a tangible way. A project needs real measures from real data. If we don't really understand software productivity factors, then we won't know how to improve our development processes. None of the current or proposed software quality and process "standards" enforce the details of the software development process, so a real software process must be built from within. But change, even when done for the right reasons, will upset the status quo and will upset established norms and behavior patterns. More importantly, it will upset the minds and perceptions of people.

We must recognize the importance of the people in the process. Improved quality and higher productivity *can* be achieved by tapping their inherent strengths. The appropriate use of metrics and statistical techniques can help us measure progress and provide support to our software development teams, while at the same time mitigating risk, lowering cost, and improving quality. Proceed with deliberate intent and create *meaningful* goals based on the benchmarking of key development processes and milestones. Your code will be happier for it. ▮.

Reference

1. See the *IEEE Standard for a Software Quality Metrics Methodology* (IEEE Std 1061-1992) for additional information with respect to essential measures.

Thomas Drake *is a management and technology consultant with Booz Allen & Hamilton, Inc. He is currently on assignment with the National Security Agency working as a senior technologist in their Software Engineering Applied Technology Center. He manages this Center's Software Quality Engineering initiatives. As part of an industry and government outreach/partnership program, he holds frequent seminars and tutorials covering code analysis, coding practice, testing, software project management, and best current practices in software development. He is currently pursuing a PhD in the information management/decision sciences and management policy tracks at the University of Maryland Baltimore County.*

NSA SEATC critical measures

C, like other high-order languages, is a communications medium similar to natural languages. It has its own grammar, vocabulary, and essential attributes. When we analyze the essential vocabulary and makeup of the language, some credible quality indicators emerge. In essence, tell-tale signatures begin to show up as the code is tracked throughout its life cycle. After analyzing 25 million lines of code, NSA has found the following critical measures to be credible quality indicators.

Critical code measures

- **Purity ratio**—The code's estimated length divided by its actual length. This is a measure of code optimization: The lower the ratio, the greater the probability that excessive code implements functionality. The higher it is above 1.00, the more optimized the code. Purity ratio also determines the extent to which impurities exist in the code. Programs that exceed their predicted length have latent impurities in them. There are six generally recognized classes of impurities: canceling of operators, ambiguous operands, synonymous operands, common subexpressions, unnecessary replacements, and unfactored expressions.
- **Volume**—An appropriate measure of program "size," especially in the presence of high operator/operand counts across larger amounts of code.
- **Development effort**—Reflects the number of mental discriminations a programmer performs to write the program/function. Our studies have shown that this number is a critical measure of code quality.
- **Cyclomatic complexity**—Measures control flow in the function based on the number of decision statements in the code. Extended cyclomatic complexity includes both decision-making statements and decision-making predicates (the number of decisions + ANDs and ORs + 1). Cyclomatic complexity is artificially inflated with case statements.
- **Functional density**—An essential indicator of the code's optimization and compactness. A function is a self-contained unit of program code designed to accomplish a particular task (an action takes place or a value is provided); at a minimum, a function is based on an input (entry point) that results in an output (exit point).

Critical coverage measures

- **Executable one-trip path count per function**—Based on segment/branch-level analysis of the code, this provides a coverage analysis and testability indicator. Programs at the extreme maximum are qualitatively untestable or require excessive testing resources.
- **Average number of segments per path**—The average number of logical links traversed between entry and exit point for all paths assigned to a function is a strong indicator of a function's relative performance. Our studies have shown a very high correlation (90+ percent) between high average branch links per path and performance degradation under operational test conditions.

A path is defined as a set of branch links (segments) between the statements in the code at the single function level, starting with an entry point and ending with an exit point. A segment corresponds to an edge and is referred to as a logical branch. The segment is a set of program statements that are executed based on the value of some logical expression or predicate in the program.

Critical summary measures

Summary measures are more process-oriented measures that scale from an individual function to all functions within a system.

- **Estimated development time per function**—A division of effort by the Stroud (*S*) index number (sliding scale). *T*^ measures how long a program should have taken to write and the estimated number of hours expended in development time per function. The Stroud number is based on the assumption that an average human makes about 18 mental discriminations per second. Exceeding this threshold is a very strong indicator that the code is volatile, less stable, and more difficult to maintain. It also reflects code that was worked on for a longer period of time by one developer over time or by multiple developers. It is also a very strong indication of the degree to which the software has atrophied. Lower numbers reflect a more stable code base.
- **Estimated error rates**—Predicted errors per KLOC is highly correlated in our studies with the degree to which the code has been "defensively" programmed and directly reflects the maturity level of both the development organization and the individual software engineer.

Maintainability and readability measures

- **Lines of code**—Should not exceed 62 lines of code per function.
- **Number of executable statements**—Should not exceed 50 executable statements per function.
- **Number of comment lines**—Should be at least 60 percent per function. This may raise some eyebrows, but NSA has a huge base of legacy code, some of which has been running for decades. Poorly commented code is both more difficult to understand and more difficult to reengineer.
- **Span of reference for variables**—Average maximum number of lines between references to each variable's assignment and use in a function. It should not be greater than 10-12 lines for various security reasons.