MANAGING SOFTWARE QUALITY THROUGHOUT THE LIFECYCLE

SOFTWARE QUALITY IS MORE THAN CHASING BUGS

Robert A. Martin and Stephen A. Morrison

1.0 MOTIVATION

How easy would it be to move all of your applications from one hardware platform to another? What if you had to change from one database management system to another, say from Oracle to Sybase? Or perhaps you might need to combine and consolidate your data processing resources with those of another organization. How would you determine which applications to abandon, which to merge, which to rewrite, and which will need no changes?

In today's rapidly changing environment, yesterday's assumptions are thrown out and "reliable" institutions may disappear tomorrow. To survive and grow in this environment organizations must be adaptable and ready for change. This is especially true in today's Information Systems arena. With ever-changing requirements from users and hardware models appearing and disappearing at breakneck speeds as well as software updates from commercial vendors constantly bombarding you, applications must be postured for growth and evolution. As software undergoes maintenance and enhancement it becomes brittle, complex, and susceptible to errors. Software quality teams can no longer afford to focus on simply removing errors: the fundamental software architectural issues of maintainability, evolvability, and portability are becoming the keys to the continued survival, as well as prosperity, of an organization.

To survive in the world today you must be better and more efficient than you were last year, in fact, better than you were yesterday. But to improve you need to know where best to devote your resources and that can not be done without a good understanding of the problems you face. To be successful in today's hectic world, organizations need to expand their definition of quality to cover issues beyond just mere defect densities and error counts.

2.0 INTRODUCTION

This paper will discuss MITRE's work in addressing lifecycle quality issues in software systems as well as the methodologies, techniques, and technologies that we have found to be useful in giving our customers an effective method for managing the quality of their applications portfolio from a lifecycle software engineering perspective. The assessment methodology we will discuss is flexible enough to allow for the analysis of systems in any language, on any platform, developed under any (or no) standards, with or without using third party commercial packages. The methodology's design is free from assumptions regarding documentation and implementation details and has broad applicability across varying architectures and development methods (traditional, 4-GL, or CASE-based).

To date, the insights provided by this methodology have been applied to issues such as measuring the cost-effectiveness of code re-use efforts, reviewing migration system candidates, and conducting consistent periodic code/project reviews of on-going development efforts. Additionally

this methodology has been used to augment the Software Engineering Institute's Software Capabilities Evaluation to obtain an assessment of the quality of the products resulting from an organization's process in addition to the quality of the process itself. In the last year over 40 Information System applications were assessed, totaling more than 21 million lines of code in a wide variety of languages. These applications had been developed on behalf of a variety of Federal agencies and Government services.

The quality assessment process produces clear and objective risk ratings of the overall quality of the software products evaluated. These ratings are comprised of specific risk drivers and risk mitigators inherent in the system artifacts under evaluation. Taken together, the ratings and list of risk drivers and risk mitigators depict a software-centric profile of the quality risks associated with each of the software products developed by an organization. This profile may then be used to guide future migration decisions, identify potential changes in an organizations development methods, or to monitor the software quality during the evolution of an organization's systems.

3.0 So How Should Quality be Defined?

There is no generally accepted and widely held definition of software quality. Everyone seems to have their own definition which supports their particular point of view and relates to the issues and concerns that they see as important. For our purposes we have defined Software Quality such that a quality system minimizes its lifecycle risks. This emphasis is in concert with the adoption of the Integrated Weapon Systems Management (IWSM) approach to systems development by Air Force Material Command. IWSM places a heavy emphasis on the lifecycle characteristics of systems and is concerned with managing the entire life of systems from a single point so that trade-offs between the design, development, and maintenance phases can be made from an integrated perspective.

In today's Information Systems organizations, software systems are expected to be modified over time, so it is of utmost concern that software support ease of modification and evolution. In this context we will want systems which minimize: the risk of introducing errors during the development and maintenance phases of the system; the risks associated with rehosting the system from one machine to another or from one version of an operating system to another; the risks associated with making enhancements to the system; the risk inherent in staff changes in the development or maintenance organization; and risks to project schedules.

3.1 A Framework for Defining Quality

There are many areas that we could examine in efforts to minimize the risks to a system. Many of those areas are already well studied and have full fledged disciplines, technologies, and examination methodologies established to attempt to control and/or positively influence their contributions to a system's quality. Specifically, the areas of requirements traceability, functional completeness, and system testability are all well established areas of study. We will focus our attention in this paper on some of the quality areas that have not received the attention these others have, but which hold an enormous opportunity for reducing the levels and types of risk in the systems we field. Specifically we will examine the four qualities areas of maintainability, evolvability, portability, and descriptiveness.

With the working definition of a quality system being one that minimizes the lifecycle risks of the system we should first explore the types of issues that we will have to examine in order to obtain a measure of system quality. In doing this we draw upon a variety of previous efforts in measuring software quality including the early Air Force efforts in assessing software quality

conducted by McCall [1] and by Bowen [2] for the Rome Air Development Center (RADC). For our purposes we are only interested in the portions of the RADC work that focused on assessing the risks to the lifecycle maintenance qualities of systems. Another source of ideas are the concepts discussed by B. A. Kitchenham from work on the ESPRIT REQUEST project [3-6], as well as other ideas discussed in the literature between 1990 and 1994 [7-11]. In developing our quality management approach, we also sought to capitalize on the existence of industry and Government open system standards and we wanted to include assessment areas that would have been impractical, if not impossible, to evaluate before the arrival of today's powerful computers and modern software reverse-engineering analysis tools and code assessment technologies.

3.2 QUALITY AREAS AND QUALITY FACTORS

The result of our integration of the above, with our own ideas, is discussed in the following paragraphs. Here we introduce seven "quality factors" which serve as a measurable foundation for the definition of the quality areas of maintainability, evolvability, portability, and descriptiveness. The factors: consistency; independence; modularity; documentation; self-descriptiveness; anomaly control; and design simplicity are less abstract than the quality areas mentioned above and will provide a better framework in which to measure the quality of a system. The relationships between these seven quality factors and the four quality areas of maintainability, evolvability, portability, and descriptiveness are shown in Figure 1 and echo the types of relationships between quality areas and quality factors that appear in the IEEE Software Quality Metrics standard [12].

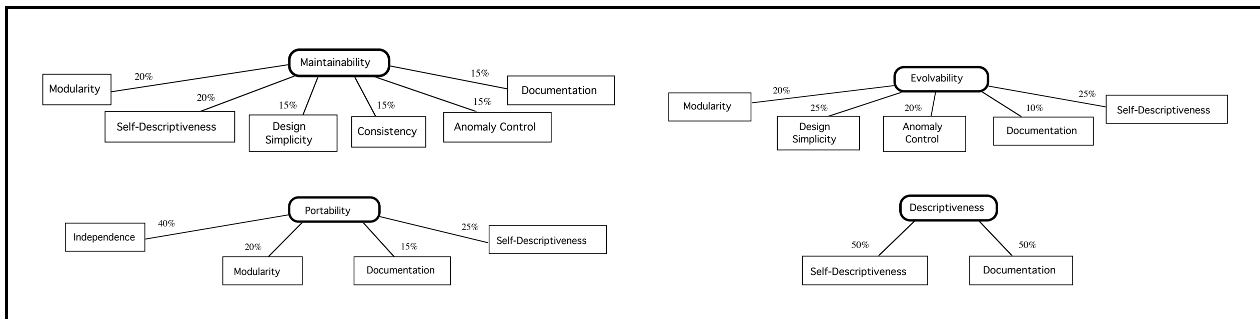

Figure 1: Quality Areas to Quality Factors Map

While a fuller definition of the seven quality factors will be made later in the paper, an introduction to the concepts that each covers is provided below to establish a context for our definitions of the four high level quality areas.

Consistency: Have the system products (code and documentation) been built with a uniform style to a documented standard?

Independence: Have ties to specific operating systems, extensions, etc. been minimized to facilitate eventual migration, evolution, and/or enhancement of the application?

Modularity: Has the code been structured into manageable segments that minimize gross coupling and simplify understanding?

Documentation: Is the accompanying documentation adequate to support maintenance, porting, enhancement and re-engineering of the application?

Self-Descriptiveness: Does the embedded documentation, naming conventions, etc. provide sufficient and succinct insight into the functioning of the code itself?

Anomaly Control: Have provisions for comprehensive error handling and exception processing been detailed and applied?

Design Simplicity: Does the code lend itself to legibility and traceability where dynamic behavior can be readily predicted from static analysis?

With these working definitions we can define maintainability, evolvability, portability, and descriptiveness in concrete terms and move on to exploring the question of how to actually go about measuring the quality of a system.

3.3 THE QUALITY OF MAINTAINABILITY

The maintainability of a software system is a broad subject ranging from architectural design issues to implementation and documentation. Under our view of maintainability, we are focused on the ease of effort in locating and fixing software failures and making minor modifications to the system. This type of maintainability is strongly influenced by the components that affect the maintainer's ability to understand the system. This includes documentation at the system and architectural level, down to the unit level data flow, as well as internal documentation and comments in the code itself. Consistency is a major concern, since standard approaches to areas of I/O and error handling provide the maintainer with a guide to how things are done, rather than requiring ad hoc discovery. Modularity is required for structuring the software in a manageable way. Simplicity of design and adequate handling of anomalies are intuitively recognized as major contributors to the amount of effort needed to maintain a portion of code.

3.4 THE QUALITY OF EVOLVABILITY

The need for evolvability is implicit in any system that requires modification. This area is concerned with the ease of effort in changing software to accommodate changes in requirements. Evolvability deals with the general question of how difficult it will be to change the capabilities of the system. This change in capability could take the form of added performance requirements (more nodes, faster, etc.), enhanced functionality (create more detailed reports), new functionality (allow the user to edit templates), or greater ease of use (add “hot keys” to menus). Simplicity of design, clean modularity, good control of errors, and informative and accurate documentation are the major determinants of the system’s evolvability.

3.5 THE QUALITY OF PORTABILITY

Concern with portability is an issue when the system may need to migrate or upgrade to another hardware platform or when operating system changes are experienced on the target hardware. The concerns here are for both the developed software and the commercial software products within the system, the language the developed software is written in, and the platforms on which the commercial software is available. We are also concerned with I/O protocols, graphics standards, the existence of any direct hardware calls, and the machine/system software specificity of any environmental parameters or control scripts of commercial software. The degree of independence is the major factor in this quality area. Modularity, from the perspective of isolating unique hardware or software features in encapsulated units or modules versus imbedding them in functional code, is also important. The extent to which the system is well documented and its use of platform unique features is described and explained is of great importance when trying to ensure the portability of a system.

3.6 THE QUALITY OF DESCRIPTIVENESS

Descriptiveness refers to both the external printed material about the system and the source code resident documentation. The concern is that the documentation be adequate to support the maintenance, porting, and enhancement activities that will occur throughout the system's life. Both high level functional descriptions and characterizations of the system as well as low level design details are needed for maintainers to understand the system's functionality as well as identify where to make changes and corrections. Modules should have standard formatted prologue sections. These sections should contain module name, version number, author, date, purpose, inputs, outputs, function of the module, assumptions, limitations and restrictions, accuracy requirements, error recovery procedures, commercial software dependencies, references, and side effects. White space and naming conventions should be used to help the legibility and understandability of the code. The judicious use of comments to highlight special features and to clarify the code's functionality is also desired.

4.0 SO HOW DO YOU MEASURE QUALITY?

For each quality factor shown in Figure 1 we have defined a mapping of that quality factor's contribution to one or more quality areas. Each quality factor is itself further defined by a set of attributes. The quality attributes are deliberately worded at the conceptual level and avoid use of subjective terms and assumptions regarding the language(s), architecture, or target platform for the system being assessed. The factor attributes are distinct, measurable questions or metrics that address the various ways the concept of the factor may be implemented in the code. For example,

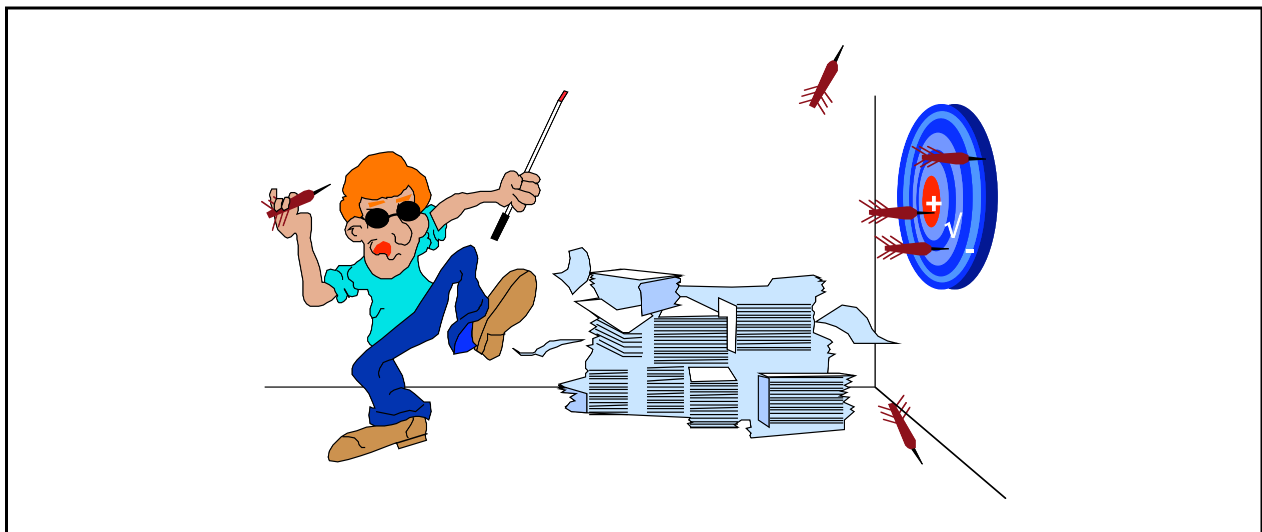

Figure 2: Of Course there are Alternate Methods of Getting a Measure of System Quality…

Figure 1 shows that the factor Self-Descriptiveness contributes to all four quality areas. As defined below, it is composed of attributes that measure the information content of the prologues and comments, use of meaningful naming conventions, white space management, etc. By enumerating a list of distinct and measurable questions or metrics that can be evaluated we can construct a structured repeatable method for measuring the quality of an application system and its supporting documentation. The following paragraphs contain discussions of the subjects that will be examined by individual quality factor attributes. The specific attribute question or metrics are contained in the Appendix.

4.1 MEASURING THE QUALITY OF CONSISTENCY

Consistency is an important factor and is a direct reflection of the policies and procedures for all aspects of the development process. Consistent software is built when a development standards document exists and development conforms with this document. Any potential issues on documentation, I/O protocols, data definition and nomenclature are addressed here. The full list of Consistency attribute questions is contained in Table 1 in the Appendix.

4.2 MEASURING THE QUALITY OF INDEPENDENCE

Measurement of independence requires attention to two broad areas: software system independence and machine independence. Here the issue is to avoid tying the system to a host environment that would make it difficult or impossible to migrate, evolve, or enhance the system. The full list of Independence attribute questions is contained in Table 2 in the Appendix.

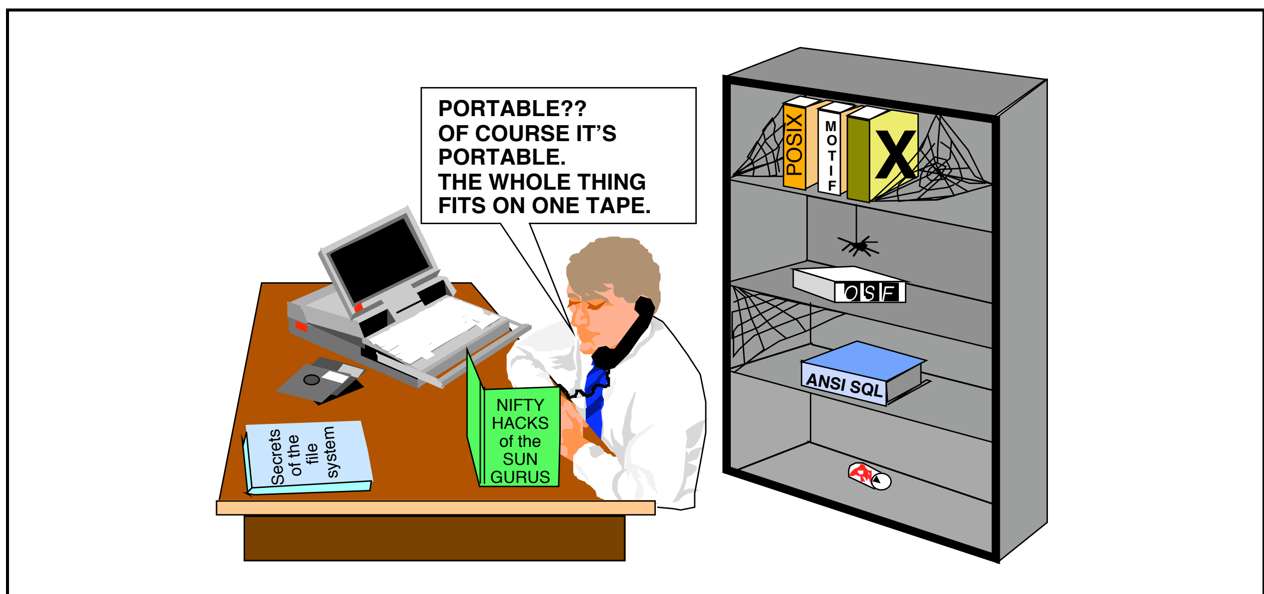

Figure 3: There are other Definitions of Portability…

4.3 MEASURING THE QUALITY OF MODULARITY

Modularity consists of several facets, each of which supports the concepts of organized separation of functions and minimizes unnoticed couplings between portions of the system. The Modularity factor is mapped to the maintainability, evolvability, and portability quality areas. The full list of Modularity attribute questions is contained in Table 3 in the Appendix.

4.4 MEASURING THE QUALITY OF DOCUMENTATION

Documentation refers to the supplemental material about the system that is external to the code. The documentation must be adequate to support the maintenance, porting, and enhancement activities that will occur throughout the system's life. High-level functional descriptions, characterizations of the system, and low-level design details are needed to support the maintainers' need to understand the system's functionality as well as identify where to make changes and corrections. Note that this differs from Self Descriptiveness, which is a quality factor for the documentation embedded within the documentation itself. The Documentation factor is mapped to the maintainability, evolvability, portability, and descriptiveness quality areas. The full list of Documentation attribute questions is contained in Table 4 in the Appendix.

4.5 MEASURING THE QUALITY OF SELF-DESCRIPTIVENESS

Modules should have standard formatted prologue sections. These sections should contain module name, version number, author, date, purpose, inputs, outputs, function, assumptions, limitations and restrictions, accuracy requirements, error recovery procedures, commercial software dependencies, references, and side effects. White space and naming conventions should be used to help the legibility and understandability of the code. The judicious use of comments to highlight special features and to clarify the code's functionality is also desired. The Self-Descriptiveness factor is mapped to the maintainability, evolvability, portability, and descriptiveness quality areas. The full list of Self-Descriptiveness attribute questions is contained in Table 5 in the Appendix.

4.6 MEASURING THE QUALITY OF ANOMALY CONTROL

Comprehensive error handling and exception processing will enhance the operation, maintenance, and modification of the system. Interfaces to other packages, systems, and commercial software need extra attention to ensure that any upgrades to these external components do not corrupt the operation of the system. The Anomaly Control factor is mapped to the maintainability and evolvability quality areas. The full list of Anomaly Control attribute questions is contained in Table 6 in the Appendix.

4.7 MEASURING THE QUALITY OF DESIGN SIMPLICITY

Design simplicity is found when all modules are structured and implemented in a manner that lends itself readily to traceability and legibility. A static analysis of the code should produce models that easily and accurately predict the code's dynamic behavior. The Design Simplicity factor is mapped to the maintainability and evolvability quality areas. The full list of Design Simplicity attribute questions is contained in Table 7 in the Appendix.

5.0 SO HOW CAN WE MEASURE QUALITY CONSISTENTLY?

Before we can structure a fully repeatable measurement instrument that is independent of the evaluator and applicable across the full gamut of languages, environments, and architectures, we need to consider the features that will provide these types of capabilities.

5.1 THE SCOPE OF QUALITY FACTOR ATTRIBUTES

For each attribute we must define the scope under which the attribute will be evaluated and define a rigid scoring criteria with evaluation guidelines.

The scope is defined as that portion of the systems being evaluated which will be examined in evaluating a particular attribute. For many attributes (especially those that can be evaluated in a

fully automated way, such as complexity metrics) the scope is all of the code, while others (the more intellectually demanding tasks) are restricted to a “representative sample.” An example of attributes that make use of sampled scopes are the questions with respect to the actual information content of embedded documentation. In these cases the scope has been defined as the seven largest, seven smallest, and seven average-sized files for each language found within the system. Thus if a system is composed of Ada, C, shell scripts, UIL, SQL, and 4-GL programs the assessment will review 126 files to assess the attributes that require a more manual assessment approach.

5.2 THE SCORING OF QUALITY FACTOR ATTRIBUTES

The evaluation guidelines used for instructing the analysts on how to assign a risk level scoring are deliberately worded at the conceptual level and avoid use of subjective terms and assumptions regarding the language(s), architecture, or target platform for the system being assessed. This allows the same framework of risk areas to be explored whether the system is on a UNIX platform using Ada and C or in an IBM MVS environment using COBOL and PL/1.

The scoring criteria and evaluation guidelines defined for each attribute use a normalized grading curve. The guidelines given to the analysts explicitly state to what level a system must address a particular feature to receive a given level of risk rating for that attribute. The rigidity of this scoring technique has been implemented and enforced to enhance the overall repeatability of the methodology while reducing the role of the analyst's subjectivity. An example of attribute questions with their scoring criteria and evaluation guidelines is shown in Figure 4. This figure also displays a dialogue box from the automated Code Assessment Toolset (CAT) which was developed to help integrate the results from individual evaluators when assessments are conducted with teams. This level of automated support is not necessary to conduct an evaluation but its use accelerates the assessment process.

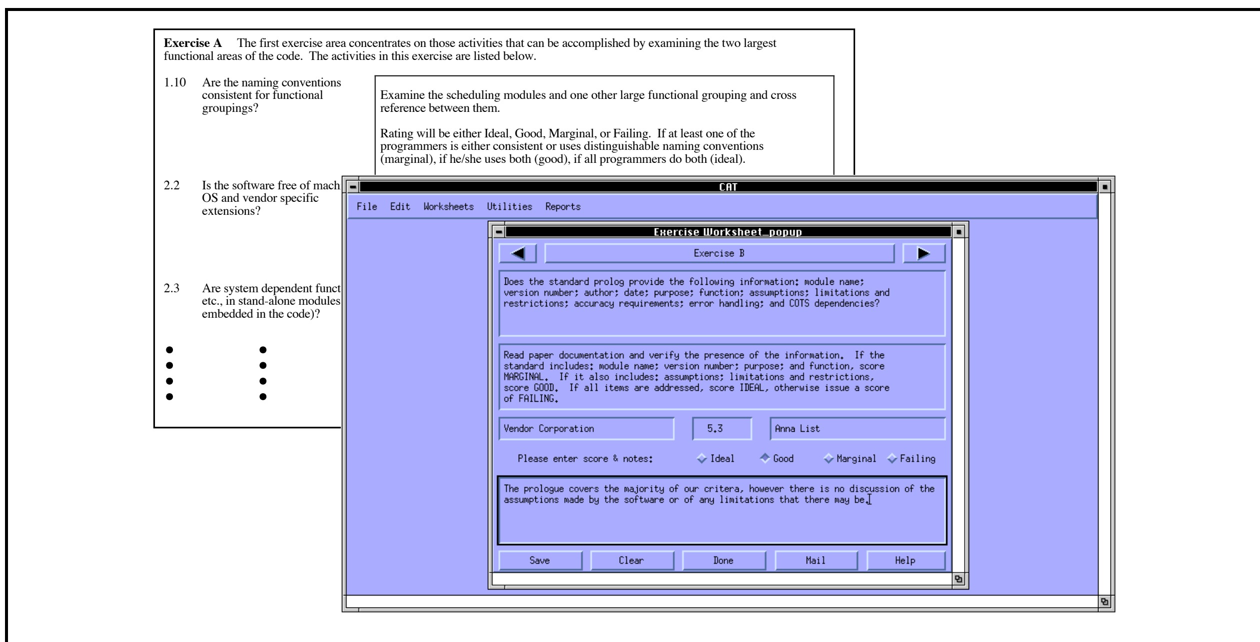

Figure 4: Example Attribute Questions with Sample Screen from the Automated Questionnaire

As stated earlier, the evaluation guidelines used for scoring are deliberately worded at the conceptual level and avoid use of subjective terms and assumptions regarding the language(s), architecture, or target platform for the system being assessed. To assess a system fairly and

address a system specific context, the evaluator needs to determine how to answer the question for a particular language within a particular operating system. To aid in this a knowledge-base is used that contains supplemental evaluation context data. This data provides the analyst with lists of known "symptoms" and analysis tools that are appropriate for a given operating environment and language mix. An on-line version of this knowledge-base is made available to analysts through the help button on the CAT questionnaire screen (see Figure 4).

6.0 So How Can We Measure Quality Efficiently?

We have already made allusions to aspects of the automated support used during assessments of systems in the previous pages of this paper. In fact, a great deal of the effort that went into the selection and wording of the factor attribute questions was aimed at maximizing the level of instrumentation and automation possible within the assessment process. While it was not always easy to create a question about a factor in a manner that allows us to use an automated method to answer it, we were able to do this for a very large percentage of the assessment questions.

6.1 Use of Automated Tools

The majority of our automation is based on programs that traverse complete directory structures while examining each of the files encountered. These programs are basically enhanced pattern matching search routines. A UNIX environment is used for our assessment and the paradigm of the UNIX "grep" functionality was chosen as the basis for these tools. To make our assessment process equally usable by those who understand UNIX and those who do not, our tools are intuitive to use and hide the platform dependencies and UNIX-ness of the tools from the analyst. A simple dialogue box appears for the analyst to select the type of search desired and to indicate within which directory the tool should search. While we have capitalized on our own knowledge of the file and directory searching capabilities of UNIX for our tools, similar capabilities could be assembled on whatever platform a group decided to use as their working environment.

6.2 Commercial Automated Tools

One may ask why we did not make use of commercial reverse-engineering and code analysis tools as the basis of our assessment automation. Early efforts attempted to use such tools but the wide variety of languages used in the systems under analysis made this infeasible. Most of the commercial analysis tools are written for one language. While the vendors have multiple versions of the same tools (one can buy either an Ada, C, FORTRAN, or COBOL analysis tool), vendors do not offer a single tool that can be used to analyze multiple languages. Another problem with the commercial analysis tool offerings is that most are focused on only main-stream languages. Script languages, GUI code, 4-GLs, SQL, and variants of languages are not supported. Since our use of software assessments is for a wide audience of customers with systems in an unknown set of languages and platforms, it would be short sighted to base our automation support on products that are constrained to only main-stream languages.

The final problem with using commercial analysis tools lies in their assumption that all of the code for a system will be available to the tool. While this is not an insurmountable problem, it does severely impact the ability to conduct assessments quickly when we are given only part of the code for a system, as is frequently the case. To use the commercial tools in that situation one must create dummy routines for each routine the code calls that is not available or one must comment out the calls themselves. This can consume considerable time and will also alter the results of some portions of the analysis itself.

6.3 OUR OWN AUTOMATED TOOLS

Since our intention is to perform assessments against an unknown numbers of systems which are using an unknown number of languages, we decided it would be more efficient to concentrate our automation efforts on the creation of easily used language independent tools to which we could adapt new languages with language profiles for any language specific capabilities needed. To date this approach has proven itself to be very successful. Currently our tools have been used on over four dozen different languages.

Eventually it is necessary to address language recognition. As mentioned above, when an assessment is arranged it is not certain what languages will be used in the system being reviewed, nor do we know how big the systems are or what percentage of the system are made up of each language. This can lead to a situation where the skills needed for a particular assessment are unknown in advance and the levels of effort are difficult to estimate. To improve the chances of successfully managing assessment efforts, a tool was developed that traverses the entire directory structure of a project, attempts to automatically recognize the types of languages it encounters, counts the numbers of files in each language, and counts the number of lines of code in each language. Any unrecognized files are also recorded and can be manually reviewed for identification purposes. The tool can be "taught" to recognize these new languages as well as count the lines of code in these new languages. By running this tool against the code of each new system we are asked to assess, we can quickly identify the languages and commercial tools used in the system as well as the level of effort that will be required for the assessment (remembering that we look at 21 files from each language used in a system for the manually intensive questions).

As of the start of this year (1995) we have used our assessment methodology and its support tools to assess over 21 million lines of code in over four dozen languages. Figure 5 indicates the major languages we have assessed. The "other" category contains over three dozen languages but only represents 4.5% of the code we have examined.

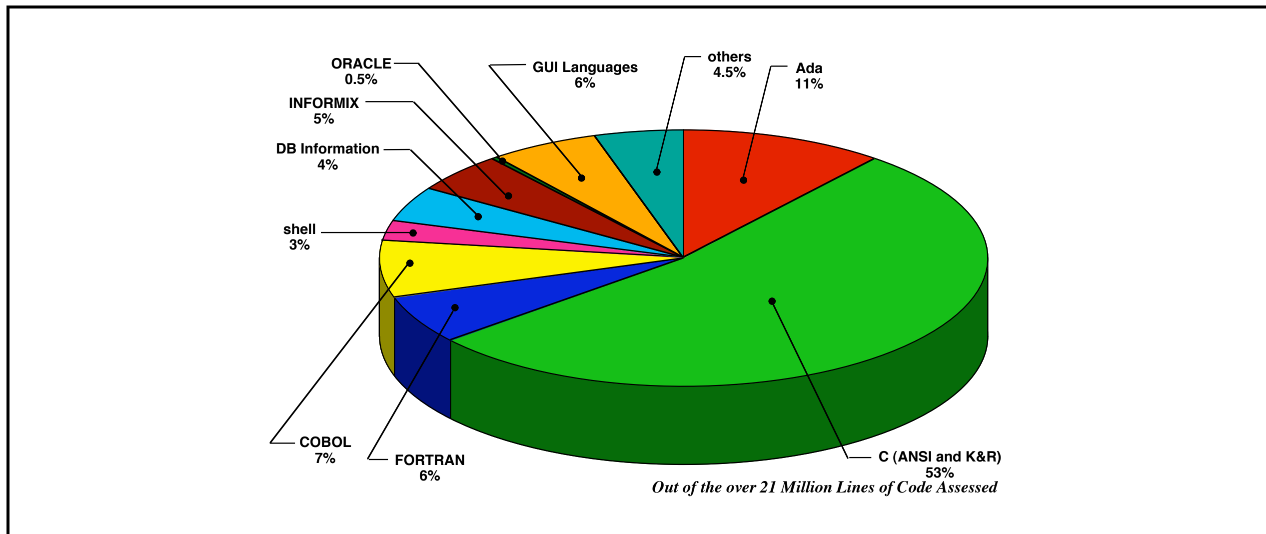

Figure 5: Distribution of Languages We Have Assessed

7.0 PRESENTING THE QUALITY ASSESSMENT RESULTS

The actual "evaluation" of an attribute consists of two parts: the assigning of a numeric score in accordance with the attribute's scoring criteria; and, the creation of an annotation detailing the specific motivation behind the score awarded. When all worksheets have been completed, their data is reduced, analyzed, and reviewed for use as the basis for the creation of a system wide quality risk profile that clearly defines the risk drivers and mitigators in the system. This allows the analysts to relate details of the implementation to high-level lifecycle engineering concepts quickly and easily and allows developers and program managers to see the long term ramifications of decisions and oversights that occurred during a system's initial development. The performance of a given system against the grading criteria can also be charted graphically at both the Quality Factor and Quality Area level as an aid in rapid identification of trends.

7.1 EXAMPLES OF ASSESSMENT RESULTS

Three specific products have been developed to convey the findings of the quality assessment to the customer. The first, which is the most important, is a risk profile that lists the major aspects of the developer's approach, software, and documentation that contribute to, or reduce, risk. These tabular reports list the major risk mitigators and risk drivers as they are revealed by the assessment, as well as any "other" items observed in the course of the examination. Samples of these tabular reports, showing some examples of the types of risks and risk mitigations found in previous assessments, and other observations, are provided below in Tables 8, 9, and 10 for illustrative purposes.

• Naming conventions used for modules and variables help understand the code's functionality.
• Good use of white space and indention.
• Modules are easily viewed at once (less than 100 LOC).
• Good functional documentation with high-level design.
• Good design documentation, showing data and control flows.
• Good top-down hierarchical structure to code.
• Modules use straight-forward algorithms in a linear fashion.
• System dependencies are to readily available COTS software.
• Code is of low complexity.
• Logic flow through individual procedures is easy to follow.
• Disciplined coding standards followed by the programmers.
• System dependenciesare isolated and all dependencies on the platform or COTS are encapsulated.

Table 8: Examples of Risk Mitigators

• Level of isolation & encapsulation of dependencies on platform & COTS varies between programmers.
• Hard coded absolute file names/paths are used.
• Hard coded variables are used when symbolic constants should have been (field declarations).
• There is some use of machine dependent data representations.
• Variables used for other than their declared purpose.
• Use of environmental variables is undocumented and inconsistently done.
• No low level control and task flows in documentation.
• No prologues for the majority of the modules.
• Inadequate indexing of documentation (cannot easily locate functionality in the code).
• No standards for handling I/O, naming conventions, or error handling are documented.
• Excessive use of global variables.
• Input error checking is not consistently applied.
• Machine generated code documentation is inconsistent with the developed code documentation.
• System tied to a proprietary language for procedural processing and data access.

Table 9: Examples of Risk Drivers

• COTS screen description files use standard X-Windows resource file formats.
• Man pages are out of date for some API's.
• The number of modules may be excessive (fewer modules may be easier to maintain).
• Proprietary language does not support data typing.
• In the vendor's proprietary language, variables are never explicitly declared. (A typo will create a variable.)

Table 10: Examples of Other Observations

The other two types of products generated are graphical depictions of the risk profiles for the quality areas and the quality factors. This information can be presented in either of two methods depending on the type of results comparison desired. A three-dimensional chart can be used to compare and contrast the risk levels of several projects against each other, or a two-dimensional presentation can be used to analyze an individual project against the standard and average for a particular quality area or factor. Samples of these graphical reports, showing some examples of the types of risk graphs generated in previous assessments, are provided below in Figures 6 and 7 for illustrative purposes.

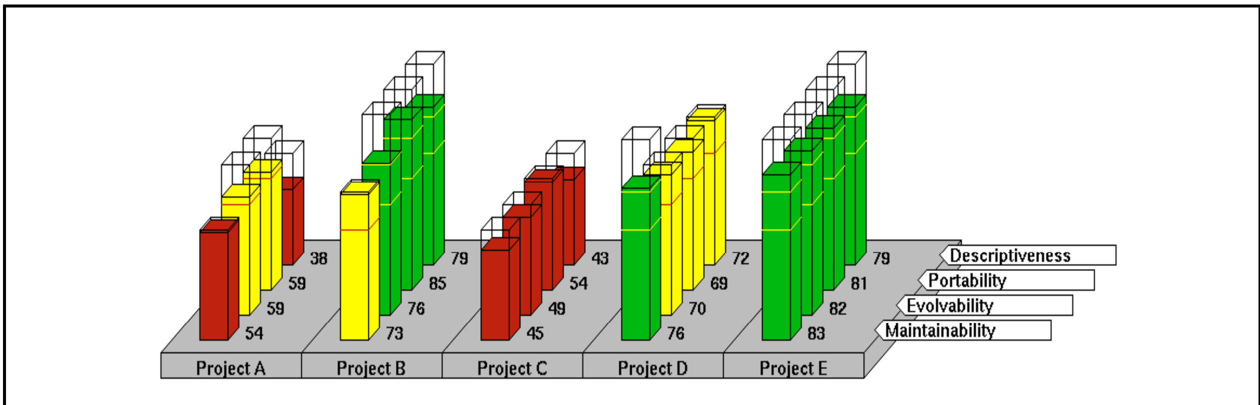

Figure 6: Sample of Multi-Project Three-Dimensional Risk Profile Graphic

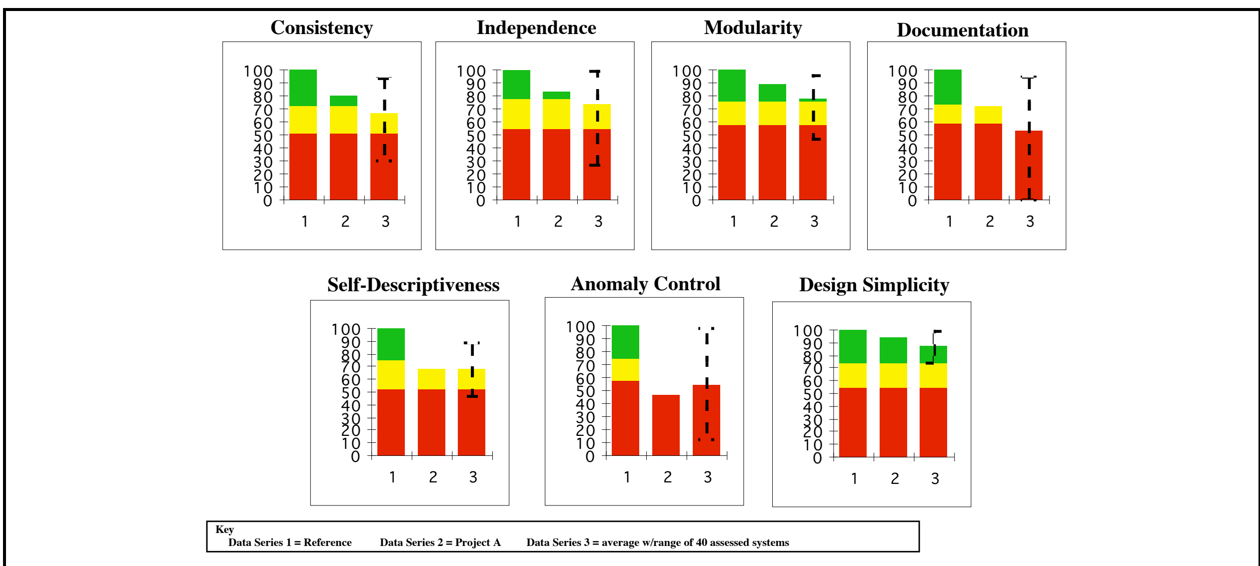

Figure 7: Sample of Single-Project Two-Dimensional Risk Profile Graphic

7.2 REACTIONS TO ASSESSMENT RESULTS

Customers have used the risk profiles in several ways, as illustrated in Figure 8. Assessments have been used as part of source selections to support the selection of a development contractor, to monitor the progress of existing contractors, and to provide technical insight into the risks of various candidate migration systems. They have also been applied to analyzing the risks of proposed re-use efforts.

Figure 8: Examples of Customers Use of Software Quality Assessments

In the course of applying software quality assessments in the three scenarios depicted above, we have had the opportunity to discuss our findings with the developers of the systems we evaluated. To date these exchanges have been positive interactions and our results (and criticisms) are well received and viewed as beneficial by the Government customer as well as by the original developers.

7.3 USERS OF THE SOFTWARE QUALITY ASSESSMENT

A list of some of the Department of Defense customers whom we have provided Software Quality Assessments includes:

- Air Force Material Command *(AFMC)*
- Electronic Systems Center *(ESC)*
- Joint Logistics Systems Center *(JLSC)*
- Human Systems Center *(HSC)*
- Air Force Operational Test and Evaluation Center *(AFOTEC)*
- Air Combat Command *(ACC)*
- Defense Information Systems Agency *(DISA)*
- Army GCCS *(AGCCS)*
- United States Transportation Command *(USTRANSCOM)*

ACKNOWLEDGMENTS

This work is funded by MITRE's Software Center Core Research effort.

REFERENCES

1. McCall, J.A., Richards, P.K., and Walters, G.F.: "Factors in software quality. Volumes I, II and II." Rome Air Development Center Reports, 1977.

2. Bowen, T.P., Wigle, G.B., and Tsai, J.: "Specification of software quality attributes. Volumes I, II and II." Rome Air Development Center Reports, 1984.

3. Kaposi, A. and Kitchenham, B.A.: “The architecture of system quality.” Software Engineering Journal, Jan 1987.

4. Kitchenham, B.A.: “Towards a constructive quality model. Parts I and II,” Software Engineering Journal, July 1987.

5. Kitchenham, B.A. and Wolker, J.G.: “A quantitative approach to monitoring software development,” Software Engineering Journal, Jan 1989.

6. Dick, R.: “Subjective Software Measurement-Tools for the Human Assessor,” ESPRIT project SCOPE report, Department of Computer Science, University of Strathclyde, Glasgow G1 1XQ.

7. Melton, A.C., Gustafson, D.A., Bieman, J.M., and Baker, A.L.: “A mathematical perspective for software measures research,” Software Engineering Journal, Sept 1990.

8. Perley, D.: “Open Systems Technology: How to Manage the Move,” Open Systems Today, 15 Nov 1993.

9. Olson, D.K.: “Developing for Multiple Platforms.,” BYTE, Feb 1994.

10. Coffee, P.: “So, have you read any good code lately?”, PC WEEK, 17 Jan 1994.

11. Rosenblum, B.D.: “Rapid Code Evaluation,” Software Development, April 1994.

12. “Standard for a Software-Quality Metrics Methodology,” IEEE Quality Metrics Standard Committee P1061, 1992.

APPENDIX

Index No.	Attribute
1.1	Is there a representation of the design in the documentation?
1.2	Is the software implemented in accordance with the representation in 1.1?
1.3	Are there consistent global, unit, and data type definitions?
1.4	Is there a definition of standard I/O handling in the documentation?
1.5	Is there a consistent implementation of external I/O protocol and format for all units?
1.6	Are data naming standards specified in the documentation?
1.7	Are naming standards consistent across languages (e.g., Structured Query Language [SQL],Graphical User Interface [GUI], Ada, C, FORTRAN)?
1.8	Are naming standards consistent across inter-process communications?
1.9	Is there a standard for function naming in the documentation?
1.10	Are the naming conventions consistent for functional groupings?
1.11	Are the naming conventions consistent for usage (e.g., I/O)?
1.12	Are the naming conventions consistent for data type (e.g., constant boolean), etc.?
1.13	Does the documentation establish accuracy requirements for all operations?
1.14	Are there quantitative accuracy requirements stated in the documentation for all I/O?
1.15	Are there quantitative accuracy requirements stated in the documentation for all constants?

Table 1. Attributes of Consistency

Index No.	Attribute
	Software System Independence
2.1	Does the software avoid all usage of specific pathnames/filenames?
2.2	Is the software free of machine, OS and vendor specific extensions?
2.3	Are system dependent functions, etc., in stand-alone modules (not embedded in the code)?
2.4	Are the languages and interface libraries selected standardized and portable? (i.e., ANSI…)
2.5	Does the software avoid the need for any unique compilation in order to run (e.g., a custom post processor to "tweak" the code to run on machine X)?
2.6	Is the generated code (i.e., GUI Builders) able to run without a specific support runtime component?
	Machine Independence
2.7	Is the data representation machine independent?
2.8	Are the commercial software components available on other platforms in the same level of functionality?

Table 2. Attributes of Independence

Index No.	Attribute
3.1	Is the structure of the design hierarchical in a top-down design within tasking threads?
3.2	Do the functional groupings of units avoid calling units outside their functional area?
3.3	Are machine dependent and I/O functions isolated and encapsulated?
3.4	Are interpreted code bodies (shell scripts and Fourth Generation Language [4-GL] scripts) protected from accidental or deliberate modification?
3.5	Do all functional procedures represent one function (one-to-one function mapping)?
3.6	Are all commercial software interfaces and APIs, other than GUI Builders, isolated and encapsulated?
3.7	Have symbolic constants been used in place of explicit ones?
3.8	Are symbolic constants defined in an isolated and centralized area?
3.9	Are all variables used exclusively for their declared purposes?
3.10	Has the code been structured to minimize coupling to globally available data?

Table 3. Attributes of Modularity

Index No.	Attribute
4.1	Is the documentation structured per the development plan?
4.2	Does the design documentation depict control flow to the CSU/CSC level?
4.3	Does the design documentation depict data flow?
4.4	Do the design documents depict the task and system initialization hierarchy/relationships?
4.5	Is the documentation adequately indexed (functionality can be easily located in the code)?
4.6	Are all inputs, process and outputs adequately defined in the documentation?
4.7	Does the documentation contain comprehensive descriptions of system/software interfaces?
4.8	Does the documentation contain comprehensive descriptions of all internal operations?
4.9	Does the documentation contain comprehensive descriptions and justification of all esoteric processing methods?
4.10	Does the documentation establish a requirement for commenting global data within a software unit to show where the data is derived, the data composition and how data is used?
4.11	Are all environmental variables and the default values clearly defined?
4.12	Does the documentation explain the high-level functionality of the system?
4.13	Does the documentation layout the functional allocation of the system to CPCIs?
4.14	Are external interfaces and systems depicted in the documentation?
4.15	Are the high-level flows of data into, out of, and through the system detailed?
4.16	Does the documentation discuss/rationalize the usage of COTS, GOTS, and OS services?

Table 4. Attributes of Documentation

Index No.	Attribute
5.1	Does the documentation specify a standard prologue?
5.2	Is a standard prologue consistently implemented?
5.3	Does the standard prologue provide the following information: module name; version number; author; date; purpose; function; assumptions; limitations and restrictions; accuracy requirements; error handling; and COTS dependencies?
5.4	Is a standard format for organizations of modules implemented consistently?
5.5	Are comments set off from code and of consistent style throughout?
5.6	Are comments accurate, and do they describe the "whats and whys?"
5.7	Do code generation tools (screen builders, DBMS query tools, etc.) produce reusable "source code" that is documented?
5.8	Are inputs, outputs, and side effects (if any) clearly detailed for each procedure?
5.9	Is any and all dead code clearly offset and the reason for its existence documented?
5.10	Has whitespace been managed for legibility and to allow identification of nesting constructs?
5.11	Are function and variable names helpful in understanding the functionality of the code?

Table 5. Attributes of Self-Descriptiveness

Index No.	Attribute
6.1	Is there a defined statement of techniques for error handling in the documentation?
6.2	Is the vendor's standard implementation of error handling consistently applied?
6.3	Is there a defined statement of techniques for tolerance of input data in the documentation?
6.4	Is the vendor's standard implementation of input data handling consistently applied?
6.5	Are tasking and rendezvous exceptions handled in an orderly fashion?

Table 6. Attributes of Anomaly Control

Index No.	Attribute
7.1	Do all modules have singular entry and exit and avoid unconditional branching internally?
7.2	Is the source code of low complexity (e.g., McCabe Cyclomatic…)?
7.3	Is the DBMS interaction properly isolated?
7.4	Does the code avoid making non-linear jumps into or out of loops?
7.5.	Does the code avoid modifying loop indices?
7.6	Do all IPCs communicate over unique channels?
7.7	Is the code segmented into procedure bodies that can be understood easily?
7.8	Is all use of self modifying code fully documented and justified?
7.9	Have all procedures been structured to avoid excessive nesting?
7.10	Have all boolean expressions been parenthesized to clarify mixed operator evaluations?
7.11	Do all boolean expressions avoid referring to both a predicate and its complement?

Table 7. Attributes of Design Simplicity

Robert A. Martin -- Stephen A. Morrison
The MITRE Corporation, MS G226
202 Burlington Road
Bedford, Mass 01730-1420
Voice: 617-271-3001
Fax: 617-271-8500
Internet: ramartin@mitre.org