

A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality

Title: A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality

Presenter: Lawrence E. Hyatt and Linda H. Rosenberg, Ph.D.

Day & Track: Wednesday, April 24, 1996. Track 5.

Keywords: Quality, Metrics, Risk

Abstract: This paper explains a Software Quality Model and then uses it as a basis for discussions on quality attributes and risks. Risks that can be determined by a metrics program are identified and classified. The software product quality attributes are defined and related to the risks. Specific quality attributes are selected based on their importance to the project and their ability to be quantified. The risks and quality attributes are used to derive a core set of metrics relating to the development process and the products, such as requirement and design documents, code and test plans. Measurements for each metric are defined and their usability and applicability discussed.

1.0 Introduction

The National Aeronautics and Space Agency (NASA) is increasingly reliant on software for the functionality of the systems it develops and uses. The Agency has recognized the importance of improving the way it develops software, and has adopted a software strategic plan to guide the improvement process. At the Goddard Space Flight Center (GSFC), the implementation of the plan has led to the establishment of the Software Assurance Technology Center (SATC), which, as part of its role of improving the quality of software at GSFC and NASA, has software metrics as one of its areas of emphasis.

NASA is not alone in attempting to improve the quality of its software; this emphasis can be seen in almost any organization where software results contribute to success, from programmable microwaves to watches to toys - quality products depend on quality software. Everyone agrees that quality is important, but few agree on what quality is. Kitchenham (1989) notes that "quality

is hard to define, impossible to measure, easy to recognize”.[7] Gilles states that quality is “transparent when presented, but easily recognized in its absence”.[5]

In this paper, we will look at a number of perspectives on software quality, and, based on a description of a typical flight project’s use of software, establish a project manager’s perspective. We will use that project description and the manager’s view to deduce requirements for a software quality model. After a review of established models for software quality, their attributes and metrics, we will present the Software Assurance Technology Center’s model for software quality, developed at GSFC to meet the needs of NASA software project and program managers. We will then discuss the use of the model and its metrics for assessing quality and risk.

2.0 Perspectives on Software Quality

Garvin concluded that “quality is a complex and multifaceted concept.” Garvin described quality from five different perspectives: the transcendental view, that sees quality as something that can be recognized, but not defined; the user view, which sees quality as fitness for the user’s purpose; the manufacturers view, which sees quality as conformance to specification; the product view, which sees quality as tied to inherent characteristics of the product; and the value-based view, which sees quality as dependent on what a customer is willing to pay for it [4].

In the software development process at GSFC, while quality is seen from all of the above views, the most important view is that of the project manager. The situation in which the software is developed and used heavily influences the project manager’s view. Figure 1 shows the process flow for building a space mission and some of the roles software plays in a space flight project. It should be noted that the software is integral to the process of developing the system. For example, the flight software and the test system software must be completed at least to some level of functionality before the integration and test of the spacecraft can be begun. Similar dependencies exist in other parts of the system.

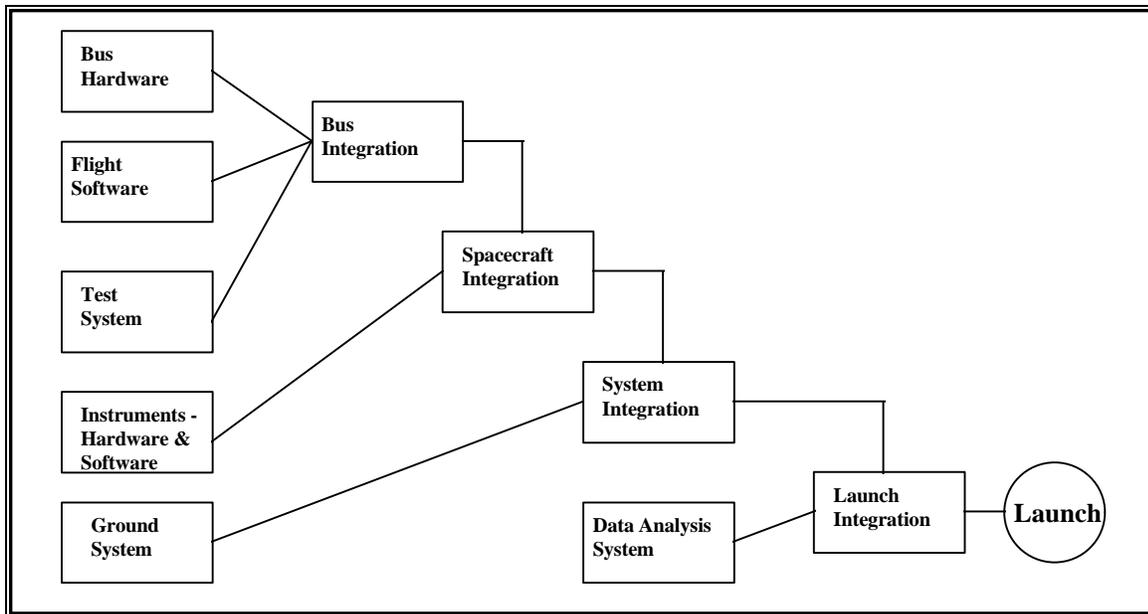


Figure 1: Process Flow for a Space Mission

Thus the project manager’s view of software quality is pragmatic and relatively simple - high quality software is software that “works well enough” to serve its intended function and is software that is “available when needed” to perform that function. The criterion of “Works Well Enough” includes satisfaction of functional, performance, and interface requirements as well as the satisfaction of typical “ility” requirements such as reliability, maintainability, reusability and correctness. The criterion of “Available When Needed” is dependent upon the software’s role in the system. Delays in availability of some software could delay the whole system and postpone the most critical date of all - the launch date.

Both of these criteria must be met for the software to be high quality, in fact, when trouble strikes, many hours are typically spent trading off activities that lead to satisfaction of one criteria for activities that would satisfy the other. In order to meet schedules, functionality may be partitioned, with highest priority requirements satisfied in an initial build, with lower priority requirements to be provided later. “Iility” requirements may also be sacrificed, with the most common tradeoff to meet schedule a reduction in documentation that will effect future maintainability and reusability, or a reduction in test time that could affect correctness and reliability. On the other hand, sometimes the software functionality and or “ility” considerations are added to in order to offset problems elsewhere in the developing system. Flight software may have to be both more functional and more reliable to decrease the need for ground system software to monitor the spacecraft.

Thus the project manager is interested in a “pragmatic” quality model and metrics program, one that will help in the successful development and operation of a specific system. Any model and associated metrics program that is to be funded by a project manager must be aimed at satisfaction of the two criteria and at the identification of risks that they will not be met.

3.0 Risk

In the Collegiate Dictionary, Webster defines risk as the possibility of loss or injury, and the degree of probability of such a loss. Barry Boehm, in his book “*Software Risk Management*”, defines risk management as a combination of risk assessment and risk control. Risk assessment includes identification, analysis, and prioritization; risk control includes management, planning, resolution and monitoring. The project risk management process may be summarized as identification of risks, ranking and prioritization of risks, and monitoring risks throughout their applicable life.

In order to develop a software quality model that is useful in the risk management process, it is necessary to identify a set of risks that is common to most projects and based on the project manager’s idea of software quality as described in Section 2.0. Given that definition, the risk areas are:

- Correctness
- Reliability
- Maintainability
- Reusability
- Schedule

The project manager might have a different prioritization of the listed risks for different segments of software that are being developed by the project (as shown in Figure 1). For example, a set of risks relevant to the flight software might be, in priority order, too many errors (low correctness), being late (schedule), and not being reusable as a basis for the next flight mission (low reusability).

For the science analysis software, however, the main risk might be low maintainability (science software changes frequently), with late delivery a secondary risk.

The needs of risk management place two additional requirements on a quality model. First, the model must support the quantification of the risks - for example, a numeric definition of the number and seriousness of the errors that can be remaining in the software before additional efforts have to be put in place to reduce the risk of low correctness. Second, the model and metrics must support the overall assessment of the project risk. That is, the risk measures must be capable of being “rolled up” into an overall measure of risk for a given software segment and for the whole project.

Once the relevant risks for a project are identified and quantified, a classification process must be defined. While it would be nice to be numeric and quantify the risk and probability of occurrence, the state of the art does not currently permit this. The SATC Model classifies risk level by the action required. It leads to the following classifications for risk levels:

LOW - Very likely to meet objectives if current trend continues. Does not need contingency plans.

MODERATE - Based on current trend, likely to meet objectives. Should have contingency plans.

HIGH - Not likely to meet objectives based on current trend. Implement contingency plans immediately.

The model does not directly predict the risk classification in most cases. Determination of the risk classification for a specific risk may involve a number of metrics, and often needs an experienced analyst to evaluate the data and assign the risk class.

The SATC has found that how the risk information is presented is important. In presentations to management, especially upper levels, a version of the “fever chart” has been found to be effective. However, since the colors used in fever charts (green, yellow, red) do not reproduce well, the symbols shown in Figure 2 are used. These symbols follow user interface guidelines using the darkest shade for the most important concept and a relevant symbol (\$) to reinforce the meaning.

		\$\$\$
Low	Moderate	High
Risk Level		

Figure 2: Risk Indicators

4.0 Software Quality Models

As defined in Section 3, project managers need a comprehensive model for the evaluation of quality and the management of risk. There are many models of software product quality that define software quality attributes. Three often used models are discussed here as examples. McCall’s model of software quality (The GE Model, 1977) incorporates 11 criteria encompassing product operation, product revision, and product transition. Boehm’s model (1978) is based on a wider range of characteristics and incorporates 19 criteria.[2] The criteria in these models are not independent; they interact with each other and often cause conflict, especially when software providers try to incorporate them into the software development process. ISO 9126 incorporates six quality goals, each goal having a large number of attributes.

The criteria and goals¹ defined in each of these models are listed in Table 1. Note that the ISO Model includes a number of criteria under its goal of maintainability.

¹ These three models and other references to software quality use the terms criteria, goals and attributes interchangeably. To avoid confusion, we will use the terminology in ISO 9126 - goal, attribute, metric.

<u>Criteria/Goals</u>	<u>McCall, 1977</u>	<u>Boehm, 1978</u>	<u>ISO 9126, 1993</u>
Correctness	X	X	maintainability
Reliability	X	X	X
Integrity	X	X	
Usability	X	X	X
Efficiency	X	X	X
Maintainability	X	X	X
Testability	X		maintainability
Interoperability	X		
Flexibility	X	X	
Reusability	X	X	
Portability	X	X	X
Clarity		X	
Modifiability		X	maintainability
Documentation		X	
Resilience		X	
Understandability		X	
Validity		X	maintainability
Functionality			X
Generality		X	
Economy		X	

Table 1: Software Quality Models

There are a number of difficulties in the direct application of any of the three models above. First, many of the criteria (goals) suggested seem, in a specific situation, to be items that should be included in the functional, performance, and interface requirements for the software as opposed to quality indicators to be measured by a metrics program. That is, the criteria “interoperability” will have a specific meaning defined in the interface requirements for the software and the software’s satisfaction of that criteria will be determined by testing. Portability needs will also be specified and tested. Efficiency will be determined by the satisfaction of performance requirements.

Second, the models are static, that is they do not describe how to project the metrics from current values to values at subsequent project milestones. The projections are needed to determine the risk of the attributes of the software satisfying the manager’s criterion for success. It is important to be able to relate software metrics to progress and to expected values at the time of delivery of the software.

Finally, the models do not give any guidance as to the use of the metrics and attributes in the identification and classification of risk.

5.0 SATC Software Quality Model

It is very difficult to convince GSFC project managers to dedicate part of a mission's budget for a metrics program, even one relating to quality and risk. The project has a long list of things that it would like to do to enhance the primary purpose of the mission (i.e., to gather and analyze scientific data), and the competition for funds is intense.

As part of its mission to improve the quality of NASA software, the SATC is assisting software managers in establishing metrics programs that meet their needs with minimal costs, and in interpreting the resulting metrics in the context of the supported projects. Using the results of these metric programs and discussions with projects as a basis, the SATC is currently defining and testing a quality model for software. Our experience indicates that the project manager's pragmatic view of software quality cannot be evaluated using only software product goals and attributes; goals and attributes for the development process through the life cycle must also be evaluated. Questions of interest are of the nature:

- Can we identify early in the development process the risks to successful completion of the software when needed?
- Will we achieve adequate correctness/reliability with the test resources and schedule currently allocated?
- What sections of the code are of the highest risk as reliability and maintainability problems?
- What constitutes code that is acceptable for reuse on other missions?

In order to convince managers to incorporate a software quality metrics program, we must show them tangible benefits, i.e., increased information about the development process and its risks, increased confidence (reduced risk) that the mission software will be usable when completed, or a real cost saving such as decreased test time. That is, in the ISO terminology, we must select a set of goals that relate to project management needs and the two questions of working well enough and being available when needed.

Following the structure of ISO 9126, the SATC model defines a set of goals that are important in the GSFC environment. The goals are then related to software product and process attributes that allow indications of the probability of success in meeting the goals. A set of metrics is chosen or developed that measure the selected attributes. The goals relate to the project manager's two questions about "working well enough" and being "available on time". From the pragmatic description of software quality, we derive four goals:

1. Requirements Quality
2. Product Quality
3. Implementation Effectivity
4. Testing Effectivity

The set of goals for software quality selected by the SATC encompass process oriented indicators of quality as well as the more traditional product oriented ones.

The model's goals must be capable of being evaluated by a set of attributes that help to define and classify risks. The attributes must be "measurable" by a set of metrics that is possible to collect within the confines of the software development process and that will yield the desired information. The goals, attributes, and metrics are explained in detail in the next section.

6.0 SATC Quality Model and Risk Evaluation

Once the goals of the software managers are defined and the attributes specified, related metrics must be selected. The SATC has defined a core metric set to give a common basis for evaluating the selected attributes of software. These metrics correlate to the quality attributes and are applicable to the goals. The metrics are applicable throughout the life cycle, including the early phases. This is very critical to project management - to assist in identifying potential problems early so they can be corrected or monitored.

The SATC Software Quality Model is shown in Table 2. The table shows all of the goals, attributes, and metrics in the model. Each goal has attributes, and each attribute has associated metrics. There are a very large number of goals, attributes, and metrics that could have been chosen for the SATC model. The specific set of goals was chosen because it supports the questions the project manager asks. The attributes were chosen because they define the goals and are related to risks. The metrics that are related to each attribute are computable from data that is collectable at reasonable cost from a software development project. The SATC had the objective that the attributes and metrics form an orthogonal set - that is, each attribute and metric appears only once in the model. The SATC also had the objective that metrics should be based on objective data collected from the processes and products and not on assessments by experts.

GOALS	ATTRIBUTES	METRICS
Requirements Quality	Ambiguity	Number of Weak Phrases
		Number of Optional Phrases
	Completeness	Number of TBDs/TBAs
	Understandability	Document Structure
		Readability Index
	Volatility	Count of changes / Count of requirements
		Life cycle stage when change is made
	Tracability	Number of software requirements not traced to system requirements
		Number of software requirements not traced to code and tests
Product(Code)Quality	Structure/Architecture	Logic complexity
		GOTO usage
		Size
	Maintainability	Correlation of complexity/size
	Reusability	Correlation of complexity/size
	Internal Documentation	Comment Percentage
	External Documentation	Readability Index
Implementation Effectivity	Resource Usage	Staff hours spent on life cycle activities
	Completion Rates	Task completions
		Planned task completions
Testing Effectivity	Correctness	Errors and criticality
		Time of finding of errors
		Time of error fixes
		Code Location of fault

Table 2: SATC Software Quality Model

6.1 Requirements Quality and Risk

The objectives for this first goal are complete, unambiguous, and understandable requirements document, the stabilization of requirements as quickly as possible, and the traceability of all requirements from their source to the software requirements document and then through design and implementation and test. The associated attributes are:

- Ambiguity - Requirements with potential multiple meanings.
- Completeness - Items left to be specified.
- Understandability - The readability of the document.
- Requirement Volatility - The rate and time within the life cycle changes are made to the requirements.
- Tracability - The tracability of the requirements upward to higher level documents and downward to code and tests.

Software requirements define the required functional requirements (what the software must do), performance requirements (how many and how fast) and interface requirements (with what, how, and with whom the software must interact). These requirements go through an iterative evolutionary process, clarifying concepts and adding additional levels of details at each iteration. If requirements are ambiguous, incomplete, or difficult to understand, the risk of an unsatisfactory final product is increased. Towards the end of the requirements phase, the requirements should stabilize with respect to additions, deletions and changes.

6.1.1 Requirements Quality Attributes and Metrics

6.1.1.1 Ambiguity

Ambiguous requirements are those that may have multiple meanings or those that leave to the developer the decision whether or not to implement a requirement. There are two metrics used to evaluate the ambiguity of the requirements document: the number of weak phrases and the number of optional phrases. Weak phrases, shown in Table 3, are not concrete, leave room for interpretation and inconsistencies and are difficult to verify. Optional phrases seem to leave it to the developer's discretion whether or not to include a feature.

Weak Phrases	Options
adequate	can
as appropriate	may
as applicable	optionally
but not limited to	
normal	
if practical	
timely	
as a minimum	

Table 3: Weak and Optional Phrases

Ambiguity is computed as the sum of the count of weak phrases and the count of optional phrases in the requirements document.

6.1.1.2 Completeness

If the requirements document is complete, it will have all of the requirements specified in adequate detail to allow design and implementation to proceed. The metrics used to evaluate completeness are a count of the TBA (to be added) and TBD (to be determined) acronyms used in the document. These acronyms are the most often used to indicate that a portion of the requirements must be supplied at some future time.

6.1.1.3 Understandability

The attribute of understandability relates to the ability of the developers to understand clearly what is met by the requirements document. The SATC is currently looking for good ways to automatically evaluate understandability. At this time we are looking at two metrics, one based on numbering structure and a second based on readability evaluations.

The width and breadth of the document's numbering structure provides an indication of extent that the requirements have been organized and the amount of detail provided. Generally, it seems that a wider and deeper numbering structure indicates a better organized document. Some research indicates that readability index of the requirements document is also an indicator of the quality. The Flesh-Kincaid Readability Index seems to have the highest correlation with actual human reaction to a requirements document.

The two metrics currently under consideration are difficult to verify as having a significant impact on the risk of using the document and therefore the attribute of understandability is not heavily used in evaluating overall requirements quality.

6.1.1.4 Volatility

A volatile requirements document is one that is changed frequently. The impact of changes to the requirements increases as one gets closer to the time when the software is to be released. The metrics for volatility are then the percentage of requirements changed in a given time period, computed as number of requirements in the changes to the document divided by the base count of requirements. The changes to be assessed come from the project configuration management system if the requirements document is under CM. If not, the changes must be located by comparisons of successive versions of the document itself.

To evaluate the number of change to the requirements, a base count of the number of requirements (a “size” measure) specified in the document is necessary. While most requirements documents use some sort of numbering system, it is readily apparent when reading a requirements document that the count of numbers is not a good count of the number of requirements, since many numbered items contain multiple requirements. Our initial work in identifying the number of requirements within a document used a count of the number of times the word “shall” appeared. This is the legally binding term that is expected in the requirement documents. The SATC research, however, has identified a list of 7 additional words or phrases that often appear in the requirement documents in place of the “shall” and usually indicate requirements.

Documents also use list structures that are introduced by some type of a continuation indicator. This use of continuations affects the count of actual requirements since they are followed by a list of requirements. Table 4 lists the imperatives and continuation phrases.

Imperatives	Continuations
shall	below:
must	as follows:
will	following:
required	listed:
responsible for	in particular:
should	support:
are to	:
are applicable	

Table 4: Imperatives and Continuations

Thus the count of requirements is the number of imperatives, plus the number of items after continuations. This count is applied to both the base document and to the changes.

6.1.1.5 Tracability

Software requirements must be derived from system requirements and be traceable to the system requirements to assure that the software developed is going to work properly in the system

setting. The software requirements must also be traceable to the implementing design and code, to ensure that they are in fact part of the software. They must be traceable to tests to ensure that they have been validated and verified. There are two metrics for tracability, the number of requirements not traced to higher level requirements and the number not traced to code and tests.

To compute these metrics requires a trace matrix which is to contain the trace information. A tool to help with upward tracing is not yet available, so only downward tracing is now done.

6.1.2 Requirements Risk

It is generally accepted that poorly written, rapidly changing requirements are a principal source of project risk (and indeed, of project failure). The SATC is working on methods to measure requirements document quality in much the same way that code quality is measured, that is, by measuring characteristics of the document itself. This measurement philosophy suffers from the problem that a well written document may still have the wrong requirements. However, it does allow an objective evaluation of the document itself. The metrics for ambiguity - the count of weak phrases and optional phrases in the requirements document do seem to result in confusion and the need to take actions to resolve the questions raised.

Volatility is an important factor in risk, and extensive measures are often taken to reduce its impact. The later in the life cycle changes are made to requirements, the resources takes to implement them. Late requirement changes may also cause a ripple effect, causing additional changes in associated areas. The earlier in the life cycle the requirements stabilize, the less the risk, as shown in Figure 3. It is the SATC experience that the traditional curve for the impact of errors can be used for the impact of requirements changes.

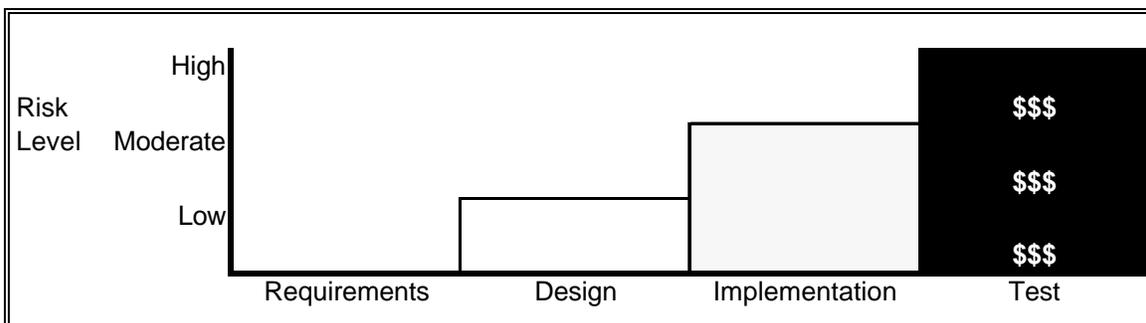


Figure 3: Phase Risk of Requirement Change

Overall assessment of requirements risk has to blend the impact of all of the requirements metrics. To reduce risk, specific requirements that are ambiguous or that contain TBAs or TBDs can be identified and revised. It should be noted that requirements risk must be measured throughout the life cycle.

6.2 Product Quality and Risk

An important objective of a software development project is to develop code and documentation that will meeting the project's "ility" requirements. The specific attributes measured are as follows.

- Structure/Architecture - The evaluation of the constructs within a module to identify possible error-prone modules and to indicate potential problems in usability and maintainability.
- Reuse - The suitability of the software for reuse in a different context or application.
- Maintainability - The suitability of the software for ease of locating and fixing a fault in the program.
- Documentation - The adequacy of internal code documentation and external documentation.

6.2.1 Product Quality Attributes and Metrics

6.2.1.1 Structure/Architecture, Reuse and Maintainability

The attributes of Structure/Architecture, Reuse and Maintainability use the same metrics for evaluation, but with different emphasis. The three areas of metrics applied here are complexity, size, and the correlation of module complexity with size.

Complexity Measurements

It is generally accepted that modules with higher complexities are more difficult to understand and have a higher probability of defects than modules with smaller values. Thus complexity has a direct impact on overall quality and specifically on maintainability and reusability. Projects developing code designed for reuse should be even more concerned with complexity since it may later be found to be more expedient to rewrite highly complex modules than to reuse them.

There are many different types of complexity measurements, for example:

- Logical (Cyclomatic) Complexity: Number of linearly independent test paths
- Data Complexity: Data types and parameter passing
- Calling Complexity: Calls to and from modules
- GOTO Usage: Count of number of GOTO statements
- Nesting Levels: Depth of nesting a conditional statement within a conditional statement.

While the SATC computes and reports all of the above, the two that are the most effective measures are Logical (cyclomatic) Complexity and GOTO use.

Logical (Cyclomatic) Complexity - The most common measure of the logic of a program is the cyclomatic complexity ($v(G)$) that was proposed by McCabe. This metric measures the number of linearly independent paths through a program, which in turn relates to the testability and maintainability of the program.

A modified or extended cyclomatic complexity is the same as cyclomatic complexity except that, in addition, each logical operator (such as *and* or *or*) within a logical statements (such as an if or while statement) adds an additional point to the complexity value. Modified or extended cyclomatic complexity will produce the highest complexity measurement for the modules.

GOTO Usage - GOTO usage generally increases the complexity of a program because it disrupts the intended or normal structured flow of the program through the application of an unconditional branching instruction. The use of large number of GOTOs is a typical cause of producing unreadable programs with “spaghetti code”. Since the GOTO structure is available (even in C and C++), and some programmers from the FORTRAN era of programming still use it with abundance, a high count of GOTOs generally indicates lower quality code.

Size Measurements

One of the oldest and most common forms of software measurement is size. There are many possibilities for representing size, but we will discuss only some common ways of counting lines. Types of line counts include lines of code, non-comment non-blank source lines of code, and executable statements.

Line of Code (LOC) - A line of code is any line of program text, regardless of the purpose or number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. All other forms of line counts are a subset of this count.

Non-comment Non-blank (Source lines of code - SLOC) - A line of code that is not a comment line or a blank line. The presence of a comment on a line with an executable statement does not preclude counting the line; the comment is ignored.

Executable Statements (EXEC) - This is a count of the number of executable statement regardless of the number of physical lines. A language delimiter, such as a semicolon in C, is usually used as an indicator.

Size of modules is a quality indicator. General industry standards suggest that 50 to 100 lines is the maximum size that any module should attain; larger modules tend to be difficult to understand and thus lower in maintainability and reusability.

Correlation of Complexity and Size

While both size and complexity are useful metrics, the correlation of the two produces even more information. The SATC has found that much of the code produced at GSFC has a linear relationship between complexity and size, such that the complexity of a “normal” module can be predicted by its size. Thus a reduction in size will reduce the complexity. The use of this correlation is important in risk determination and will be discussed in section 6.2.2

6.2.1.2 Internal Documentation

Documentation is the description of the content of the code. Documentation can be in the forms of manuals external to the code, or comments within the code itself. The metric recommended for this attribute is the comments within the code since external documentation is often not available or not used when changing source code. Comments within the code are readily available and the primary source of information when making changes to the code. Many organizations have specific guidelines for what should be contained in the commented header section of each code module including dates and changes that allows for segment error tracking. Commenting within the actual body of the code is often not specified, but critical for understanding.

In counting the number of comments within the body of the code, both on-line and stand-alone comments must be counted since the type of commenting often depends on the preference of the programmer. On-line comments are contained on a line with an executable statement and many code analysis tools neglect to count them, skewing the comment percentage. Stand-alone comments are on a line by themselves, such as in header code and are very easy to identify. Some code analysis tools also count comment words to assist in identifying blank comment lines that serve as spacing but add nothing to the understanding of the program.

Comment percentage is calculated by dividing the number of comments by the total lines of code less the blank lines ($\# \text{ comments} / (\text{total lines of code} - \text{blank lines})$).

6.2.1.3 External Documentation

At this time, the only measure of quality of external documentation used by the SATC is the readability indices, as discussed under requirements.

6.2.2 Product Quality Risk

While poor quality of any product is a risk to the specific objectives of the project, some of the best measures of risk come from correlations of the base metrics. The diagram in Figure 4 is to be overlaid on a scatter plot of all of the code modules in a program or segment of the system software. The diagram, when completed, identifies the risks to correctness and, especially, maintainability and reusability based on size and complexity. The risk assigned to the areas in the diagram are based on the SATC’s experience. The diagram’s values are used to show risk for maintainability by the number of modules in the higher risk areas. Lower values for complexity

and size would be used to redefine the risk areas for reusability since the candidates for reusability have both upper and lower guidelines.

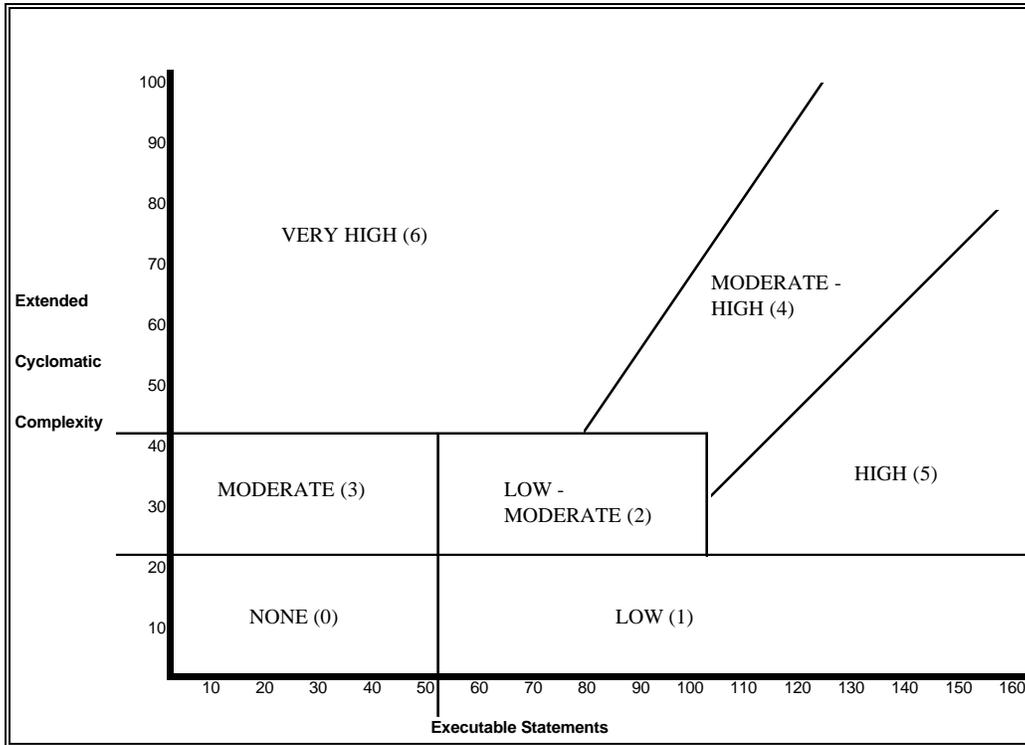


Figure 4: Module risk areas

6.3 Implementation Effectivity and Risk

The objective of implementation effectivity is to maximize the effectiveness of resources within the project scheduled activities. The attributes of this goal are:

- Resource use - The extent resource usage correlates to the appropriate phase of the project.
- Completion Rates - progress made in completing items such as peer reviews, or turnover of completed modules to CM.

6.3.1 Implementation Effectivity Attributes and Metrics

6.3.1.1 Resource Use

The resource metric recommended is personnel hours devoted to a set of life cycle activities. The metrics use resource data as a measure of the types of activities that are being done at any specific point in the life cycle. To the extent that those activities do not match the expected or planned activities, an element of risk may have been identified. For example, work on prior phase activities during the current phase is a risk indicator. The measures collected are staff hours spent on a list of activities that is tailored for the project (and may have to change as the project progresses). Key activities that should be measured include: requirements analysis, coding, testing, corrective action, training, etc.

6.3.1.2 Completion Rates

Completion rates of tasks and product components are a good indicator of risk that a project will or will not be able to provide products on the needed schedule. Completion rates can be measured, and if a detailed schedule is available, completions can be compared to the schedule to provide planned versus actual charts. Completion rates and planned versus actual charts are usually thought of as the responsibility of the administrative support staff in a project rather than of the metrics program. If a good schedule and completion measurement program is underway, that division of labor can be maintained. However, a complete picture should be available to the project manager, and, for example, charts and measures of completion of peer reviews, the progress of modules through unit test and the completion rate of tests should be kept and displayed. This, combined with the resource measures, gives a good picture of what is going on and the impact of the carryover of prior phase activities.

It should be emphasized that resource and completion rate data must not be allowed to be used to measure productivity of development teams or individuals. If individuals feel they are being measured by the metrics, the whole process will collapse. It has been demonstrated over and over again that if the development staff feels that individuals are being measured the staff will cease to cooperate with the metrics program, and indeed are likely to actively sabotage it.

6.3.2 Implementation Effectivity Risk

An example of the risks that can be determined from the metrics of resource use is based on the appropriateness of the tasks to which resources are being applied at a given time during the life cycle. If significant effort is being expended on requirements activities during the design phase (or worse, during the implementation phase), there is significant risk that the project will not be able to meet its schedule objective.

As an example of the use of completion rate data in determining risks, the completion rate of tests can be used to estimate the risk of competing all tests on time. The risk is measured by use of test report data. Table 5 is an example of data that might be compiled from the test data and used for projecting testing results for the next 3 months of testing. Using this information, a project manager might conclude that if all tests must be passed by the end of the eighth month, this project is at high risk.

Test Month	# Tests Run	# Test Passed	# Tests Remain	# Req Tested	# Req Passed	# Req Remain	# Mod Tested	# Mod Passed	# Mod Remain
1	5	4	1	15	12	3	34	29	5
2	8	6	2	25	17	8	50	37	13
3	12	6	6	31	17	14	68	40	28
4	15	7	8	51	21	30	102	39	63
5	21	11	10	62	30	32	138	61	77
6	26	13	13	70	32	38	168	69	99
7	32	16	16	97	45	52	233	96	137
8	35	18	17	95	43	52	220	91	129
Projections									

Table 5: Sample Test Data

6.4 Testing Effectivity and Risk

The objectives for effective testing is to locate and repair faults in the software, to identify error-prone software, and to complete testing on schedule with sufficient faults found and repaired that the software will operate “well enough” when it is put into operation. The attribute measured is Correctness.

Once code has been generated and gone through Unit testing, formal testing - System, Integration, and Acceptance Testing - begins. This testing may include the end users, especially in Acceptance Testing. During formal testing, all software modules are integrated into a cohesive whole and a series of system integration and validation tests are conducted. The purpose of this testing is to find errors which normally result from unanticipated interactions between subsystems and components. It is also concerned with validating that the overall system provides functions specified in the requirements and that the dynamic characteristics of the system match those required by the system procurer. Formal System testing is actually a series of different tests whose primary purpose is to fully exercise the system.

6.4.1 Testing Effectivity Attributes and Metrics

6.4.1.1 Correctness

Correctness is defined as the extent the code fulfills specifications. One implication is that the software must be error free. Errors are located by inspections and other peer reviews, by unit testing, and by integration and system testing. Most GSFC projects do not use inspections, and do not record errors found during unit testing. This attribute can only be measured during system level tests and therefore is used as part of the test effectiveness goal.

The correctness metrics are shown in Table 6.

Date detected
Date closed
Criticality
Test #
Origin of defect
Code affected by closure

Table 6: Error Information

A cumulative error distribution, as shown in Figure 5, is typical of the rate of finding errors in a test situation.

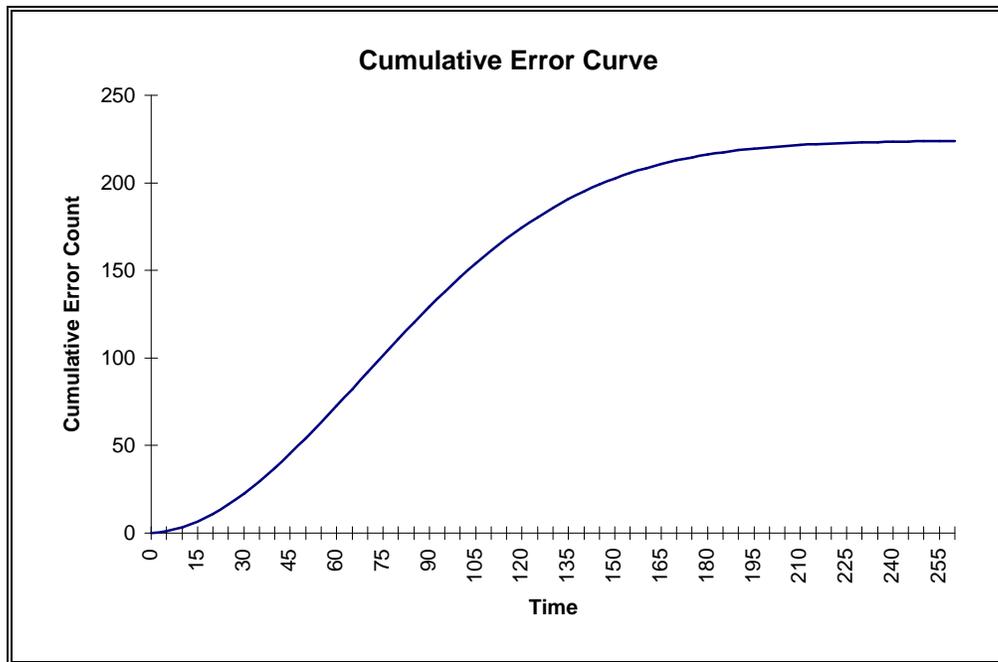


Figure 5: Typical Rate for Locating Errors

The SATC has developed a model that uses the typical error curve equation and does a curve fit with the error data from testing. After the projections have settled down (about a third of the way through the test process), the fitted curve can be used to predict the total number of errors that will be found and the schedule resource needed to find them. We have also had some success in predicting the total number of errors by criticality category.

Correctness is reported as the number of errors found and, after the model has settled down, the expected total number of errors.

6.4.2 Testing Effectivity Risk

The metrics for correctness lend themselves to a number of risk projections. If the project has quantified the percentage of the errors that are to be found before the software is accepted, the model can be used to project when that will happen, and the time at which it will occur. If the time is significantly after the time at which the schedule requires the software to be available, then risk is high.

Another risk determination can be done by looking at the code modules or sub-systems in which the errors occur. Segments of code with more than the average number of errors may well need another look to see if re-engineering should be considered. See section 7.0 for more on this subject. Additionally, the time to fix errors should be tracked. If this metric is increasing, then a threat to correctness and to schedule exists and should be pointed out to the project.

Finally, the location of faults that caused high criticality errors should be tracked. A concentration of them in a segment of code indicates a risk that the requirements are not well understood, or that the design is not suitable.

7.0 Multi - Metric Risk Evaluations

In addition to the risks that can be determined and classified based on the metrics collected for a specific goal, some risk determinations can be improved by consideration of metrics from another goal. For example, high requirements volatility can not only be measured by a count of requirements changes, but by the amount of staff resources being spent on requirements after the end of the requirements analysis phase of the life cycle. This data is available as a metric supporting Goal 3. An interesting metric is the amount of effort spent on the average change, which can be computed by dividing the total changes in some period into the total resources spent on requirements analysis during the period.

Another multiple metric risk determination can be done by using the risk ranking for modules based on their size and complexity, from Figure 4, and adding consideration of the number and criticality of errors found in the modules. As is obvious, modules with high values in all areas are those that should be examined for re-engineering.

The SATC has found that one of the parameters in the curve fit to the cumulative error curve (see Figure 5) is related to test resources. Our current interpretation of this resource is staff resources, and have gotten good results by constraining the solution to the error curve to a measure of staff, gained from the resource metrics in Goal 2. Given this, new error curve solutions can be worked out if the project proposes to change the test resources (usually to try to speed up a test process that is falling behind).

If a project is in the test phase, and metrics from product quality and test effectivity are available, the can be used to classify the remaining project risk. Using the risk depiction method shown in Figure 2, the risk presentation to the project might look like Figure 6.

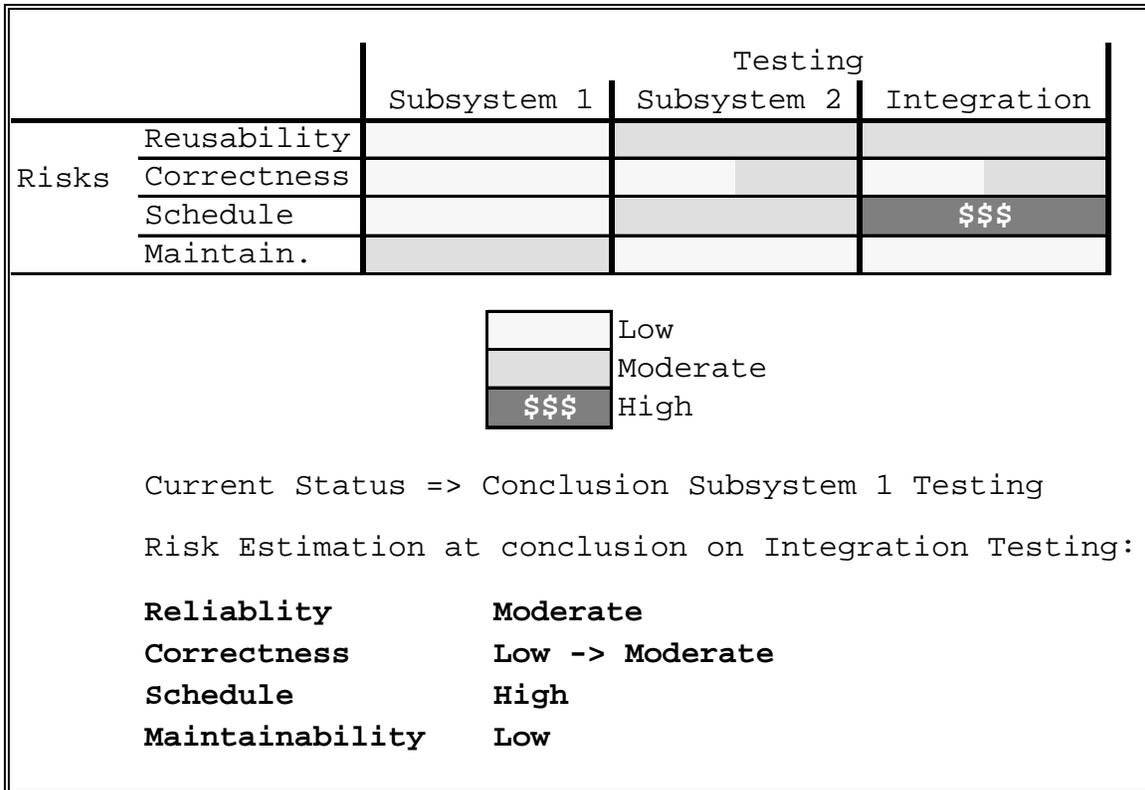


Figure 6: Goal Presentation

This aspect of the use of metrics is currently being examined by the SATC. We expect some significant new risk determinations to arise from this combined use of the metrics.

8.0 Summary

The Software Assurance Technology Center has applied some of the concepts from the theoretical models of software quality to develop a unique model that fits the needs of project managers in the NASA and GSFC environment.

- The model is dynamic, not static, in the fact that it allows the production of multiple snapshots of project status across the development. The data is used to make projections about specific project risks at project milestones.
- The model uses a broad range of measures, since it contains goals, attributes, and metrics for both software products and development processes.

- The model is comprehensive, starting with specifying goals through presenting the results. It is also applicable across the development life cycle by adapting the goals to the life cycle phase based on the project's specific objectives. The Model's metrics are derived based on aspects of the attributes that answer questions of the project managers. The model includes analysis guidelines for the data collected.

9.0 Future Work

Currently, working with the SATC, software managers and quality assurance engineers are starting to budget for data collection and software metric analysis. Significant effort is being spent on methods to use the metrics to forecast the values of the selected goals and attributes forward to project milestones such as delivery of the software.

Metrics introduced by the SATC, especially those relating to requirements quality need to be validated, and a set of higher level design quality metrics is needed to supplement those done at the code level. Our long term objective is to be able to establish a numerical metric scale for assessment of all of the metrics that affect software quality.

References And Related Material

- [1] Banker, R., Datar, S., Kemerer, C., Zweig, D., “Software Complexity and Maintenance Costs”, *Communications of the ACM*, November, 1993, pp. 81-93.
- [2] Boehm, B., *Software Risk Management*, IEEE Computer Society Press, CA, 1989.
- [3] Dromey, R.Geoff, “A Model for Software Product Quality”, *IEEE Transactions on Software Engineering*, February, 1995, pp. 146-162.
- [4] Garvin, D., “What Does ‘Product Quality’ Really Mean?” *Sloan Management Review*, Fall 1984, pp 25-45
- [5] Gillies, Alan C., *Software Quality, Theory and Management*, Chapman & Hall, 1992, pp. 19-40.
- [6] *IEEE Standard for a Software Quality Metrics Methodology*, IEEE Std. 1061-1992, 1992.
- [7] Kitchenham, B., and Pfleeger, S., “Software Quality: The Elusive Target” *IEEE Software*, January 1996, pp 12-21
- [8] Kitchenham, B., Walker, J., “A Quantitative Approach to Monitoring Software Development, *Software Engineering Journal*, January, 1989.

Lawrence E. Hyatt

Mr. Hyatt is a member of the Systems Reliability and Safety Office at NASA's Goddard Space Flight Center where he is responsible for the development of software implementation policy and requirements. He founded and leads the Software Assurance Technology Center, which is dedicated to making measured improvements in software developed for GSFC and NASA.

Mr. Hyatt has over 35 years experience in software development and assurance, 25 with the government at GSFC and at NOAA. Early in his career, while with IBM Federal Systems Division, he managed the contract support staff that developed science data analysis software for GSFC space scientists. He then moved to GSFC, where he was responsible for the installation and management of the first large scale IBM System 360 at GSFC. At NOAA, he was awarded the Department of Commerce Silver Medal for his management of the development of the science ground system for the first TIROS-N Spacecraft. He then headed the Satellite Service Applications Division, which developed and implemented new uses for meteorological satellite data in weather forecasting. Moving back to NASA/GSFC, Mr. Hyatt developed GSFC's initial programs and policies in software assurance and was active in the development of similar programs for wider agency use. For this he was awarded the NASA Exceptional Service Medal in 1990.

He founded the SATC in 1992 as a center of excellence in software assurance. The SATC carries on a program of research and development in software assurance, develops software assurance guidance and standards, and assists GSFC and NASA software development projects and organizations in improving software processes and products.

Contact: Larry Hyatt
 GSFC
 Code 302, Bld 6
 Greenbelt, MD 20771

 301-286-7475 (voice)
 LHyatt@gsfc.nasa.gov

Linda H. Rosenberg, Ph.D.

Dr. Rosenberg is an Engineering Section Head at Unisys Government Systems in Lanham, MD. She is contracted to manage the Software Assurance Technology Center (SATC) through the System Reliability and Safety Office in the Flight Assurance Division at Goddard Space Flight Center, NASA, in Greenbelt, MD. The SATC has four primary responsibilities: Metrics, Standards and Guidance, IV&V, and Outreach programs. Although she oversees all work areas of the SATC, Dr. Rosenberg's area of expertise is metrics. She is responsible for overseeing metric programs to establish a basis for numerical guidelines and standards for software developed at NASA, to investigate the role of metrics in risk assessment and management of software projects, and to work with project managers to use metrics in the evaluation of the quality of their software. As part of the SATC outreach program, Dr. Rosenberg has presented metrics/quality assurance tutorials at GSFC, and IEEE and ACM conferences.

Immediately prior to this assignment, Dr. Rosenberg was an Assistant Professor in the Mathematics/Computer Science Department at Goucher College in Towson, MD. Her responsibilities included the development of upper level computer science courses in accordance with the recommendations of the ACM/IEEE-CS Joint Curriculum Task Force, and the advisor for computer science majors.

Dr. Rosenberg's work has encompassed many areas of Software Engineering. In addition to metrics, she has worked in the areas of hypertext, specification languages, and user interfaces. Dr. Rosenberg holds a Ph.D. in Computer Science from the University of Maryland, an M.E.S. in Computer Science from Loyola College, and a B.S. in Mathematics from Towson State University. She is a member of Electrical and Electronic Engineers (IEEE), the IEEE Computer Society, the Association for Computing Machinery (ACM) and Upsilon Pi Epsilon.

Contact: Dr. Linda Rosenberg
GSFC
Code 300.1, Bld 6
Greenbelt, MD 20771

301-286-0087 (voice)
linda.rosenberg@gsfc.nasa.gov