N92-19427

Pragmatic Quality Metrics For Evolutionary Software Development Models

63120
P-31

Walker Royce
TRW Space and Defense Sector
Redondo Beach California

T 60126

ABSTRACT

Due to the large number of product, project and people parameters which impact large custom software development efforts, measurement of software product quality is a complex undertaking. Furthermore, the absolute perspective from which quality is measured (customer satisfaction) is intangible. While we probably can't say what the absolute quality of a software product is, we can determine the relative quality, the adequacy of this quality with respect to pragmatic considerations, and identify good and bad trends during development. While no two software engineers will ever agree on an optimum definition of software quality, they will agree that the most important perspective of software quality is its ease of change. We can call this flexibility, adaptability or some other vague term, but the critical characteristic of software is that it is *soft*. The easier the product is to modify, the easier it is to achieve any other software quality perspective.

This paper presents objective quality metrics derived from consistent lifecycle perspectives of *rework* which, when used in concert with an evolutionary development approach, can provide useful insight to produce better quality per unit cost/schedule or to achieve adequate quality more efficiently. The usefulness of these metrics is evaluated by applying them to a large, real world, Ada project (CCPDS-R).

These measures can be automated, consistent, and easy to use. Along with subjective interpretation to account for the lifecycle context, objective insight into product quality can be achieved early where correction or improvement can be instigated more efficiently.

Index Terms- Evolutionary Development, Software Quality Metrics, Ada, Maintainability, Process Improvement.

BACKGROUND

There have been many attempts to define measures of software quality in the past 20 years. For many reasons, none of these has caught on as accepted practice in the software industry. [2] discusses many of the problems and tradeoffs associated with defining and measuring software quality. One of the recurring themes in this work was the need for subjectivity and expensive human resources in both the collection and interpretation of quality metrics. Furthermore, the concept of a technology independent set of metrics, although an acknowledged desire, was not well understood. [8] provides an excellent discussion of the need for objective, measurable software quality metrics which remain technology independent. [9] defines a complete company metrics program with actual data that provides some valuable experience and lessons learned. [10] describes the most current motivation for measuring software quality, process improvement.

After three years of successful software development on the Command Center Processing and Display System - Replacement (CCPDS-R) project using modern Ada software engineering techniques ([12], [13] and [15]), TRW has derived a subset of software quality metrics which are measurable, objective, and useful in providing a basis for improving downstream quality of products and processes. One of the problems with typical government contracted systems like CCPDS-R is that most are one of a kind projects. This characteristic provides added complexity to measurement since the experience may be only partially useful between different project domains. The metrics presented herein have been formulated to be as useful as possible while remaining relatively domain independent so that comparisons between different projects are possible. This is not as simple as it sounds and the literature on software quality metrics reinforces this experience. After many iterations, the data presented herein has demonstrated objective and valuable insight in its application to CCPDS-R and it provides a credible basis from which better metrics can be derived.

Software Quality Metrics Objectives. Software quality metrics should be simple, easy to use, and hard to misuse. They should be useful to project management, stimulate continuous improvement of our development process, and low cost to administer consistently across different projects.

Usefulness. Conventional testing techniques exist for assessing the *functionality*, *reliability* and *performance* of a software product, however, there are no accepted methods for assessing its flexibility (*modularity, changeability, or maintainability*). While there are many other perspectives of quality (e.g., portability, interoperability, etc.), our experience in executing an evolutionary development process has demonstrated that its flexibility aspects are the most important. The easier the product is to modify, the easier it is to achieve any other software quality perspective except perhaps performance. The tradeoff between flexibility and performance is highly dependent on the application domain as well as many other architectural issues and for the purposes of this discussion we will assume that performance is achieved through proper hardware selection and that the project is prioritized "software first". A project which is prioritized more towards performance (i.e., 1750A flight program), may not interpret these metrics in the same fashion as a project prioritized towards continuous lifecycle modification (i.e., ground based C^3 System). This paper will attempt to provide useful, objective definitions for modularity, changeability and maintainability. The intent of this metrics program is to provide a mechanism for quantifying both end-product quality as well as in-progress development trends toward achieving that quality.

Development Language. Ada has proven to support increased quality and the evolutionary process model in large software development efforts. Furthermore, Ada appears to be the language of choice for the majority of current and future large government projects. While this paper assumes that Ada is the language for design and implementation of software development projects which use these software quality metrics, it should be straightforward to adapt this approach to other languages through a suitable redefinition of a Source Line of Code (SLOC).

Development Approach. An evolutionary development approach as prescribed in the Ada Process Model [12] is necessary to maximise the usefulness of these metrics across a broader range of the life cycle. The metrics are derived from controlled configuration baselines. Therefore, an approach with early incremental baselines will see an increased benefit. As a prerequisite to understanding the derivation of the software quality metrics, the following section provides an overview of the Ada Process Model employed on CCPDS-R.

Ada PROCESS MODEL

An Evolutionary Process Model is fundamental to this approach for Software Quality Assessment. Without tangible intermediate products, software quality assessment would be ineffective and inaccurate. Conventional experience has repeatedly seen projects sequence through highly successful preliminary and critical design phases (as perceived by conventional Design Review assessment of design quality) only to have the true quality problems surface in the integration and test phases with little or no time for proper resolution. An Evolutionary Process Model provides a systematic approach for achieving early insight into product quality and a uniform lifecycle measure for its assessment. It also avoids the inevitable degradations in quality due to late breakage and rapid fixes which are shoehorned into the product without adequate software engineering.

TRW's Ada Process Model is, in simplest terms, a uniform application of incremental Ada product evolution coupled with a demonstration-based approach to design review for continuous and insightful thread testing and risk management. The techniques employed within this process are derived from the philosophy of the Spiral Model [7] with emphasis on an evolutionary design approach. The use of Ada as the life cycle language for design evolution provides the vehicle for uniformity and provides a basis for consistent software progress and quality metrics.

TRW's Ada Process Model recognizes that all large, complex software systems will suffer from design breakage due to early unknowns. It strives to accelerate the resolution of unknowns and correction of design flaws in a systematic fashion which permits prioritized management of risks. *The dominant mechanism for achieving this goal is a disciplined approach to incremental development.* The key strategies inherent in this approach are directly aimed at the three main contributors to software diseconomy of scale: minimising the overhead and inaccuracy of interpersonal communications, eliminating rework, and converging requirements stability as quickly as possible in the lifecycle. These objectives are achieved by:

1. requiring continuous and early convergence of individual solutions in a homogeneous life cycle language (Ada).

2. eliminating ambiguities and unknowns in the problem statement and the evolving solution as rapidly as practical through prioritized development of tangible increments of capability.

Although many of the disciplines and techniques presented herein can be applied to non-Ada projects, the expressiveness of Ada as a design and implementation language and support for partial implementation (abstraction) provide a strong platform for creating a uniform approach.

Many of the Ada Process Model strategies (summarized in Figure 1) have been attempted, in part, on other software development efforts; however, there are fundamental differences in this approach compared to conventional software development models.

Process Model Strategy		Conventional Counterpart
Uniform Ada Lifecycle Representation	$\Longrightarrow$	PDL/HOL
Incremental Development	$\Longrightarrow$	Monolithic Development
Design Integration	$\Longrightarrow$	Integration and Test
Demonstration Based Design Review	$\Longrightarrow$	Documentation Based Design Review
Total Quality Management	$\Longrightarrow$	Quality by Inspection

Figure 1: New Techniques vs. Conventional Techniques

Uniform Ada Lifecycle Representation. The primary innovation in the Ada Process Model is the use of a single language for the entire software lifecycle, including, to some degree, the requirements phase. All of the remaining techniques rely on the ability to equate design with code so that the only variable during development is the level of abstraction. This provides two essential benefits:

1. *The ability to quantify units of software (design/development/test) work in* **one dimension**, *Source Lines of Code (SLOC).* While it is certainly true that SLOC is not a perfect absolute measure of software, with consistent counting rules, it has proven to be the best normalized measure and does provide an objective, consistent basis for assessing relative trends across the project life cycle.

2. *A formal syntax and semantics for lifecycle representation with automated verification by an Ada compiler.* Ada compilation does not provide complete verification of a component. It does go a long way, however, in verifying configuration consistency, and ensuring a standard, unambiguous representation.

Incremental Development. Although risk management through incremental development is emphasized as a key strategy of the Ada Process Model, it was (or always should have been) a key part of most conventional models. Without a uniform lifecycle language as a vehicle for incremental design/code/test, conventional implementations of incremental development were difficult to manage. This management is simplified by the integrated techniques of the Ada Process Model.

Design Integration. In this discussion, we will take a simple minded view of "design" as the structural implementation or partitioning of software components (in terms of function and performance) and definition of their interfaces. At the highest level of design we could be talking about conventional requirements definition, at the lowest level, we are talking about conventional detailed design and coding. Implementation is then the development of these components to meet their interfaces while providing the necessary functional performance. *Regardless of level, the activity being performed is Ada coding.* Top level design means coding the top level components (Ada main programs, task executives, global types, global objects, top-level library units , etc.). Lower level design means coding the lower level program unit specifications and bodies.

The postponement of all coding until after CDR in conventional software development approaches also postponed the primary indicator of design quality: integrability of the interfaces. The Ada Process Model requires the early development of a Software Architecture Skeleton (SAS) as a vehicle for early

interface definition. The SAS essentially corresponds to coding the top level components and their interfaces, compiling them, and providing adequate drivers/stubs so that they can be executed. This early development forces early baselining of the software interfaces to best effect smooth evolution, early evaluation of design quality and avoidance of downstream breakage. In this process, we have made integration a design activity rather than a test activity. To a large degree, the Ada language forces integration through its library rules and consistency of compiled components. It also supports the concept of separating structural definition (specifications) from runtime function (bodies). The Ada Process Model expands this concept further by requiring structural design (SAS) prior to runtime function (executable threads). Demonstrations provide a forcing function for broader runtime integration to augment the compile time integration enforced by the Ada language.

Demonstration Based Design Review. Many conventional projects built demonstrations or benchmarks of standalone design issues (e.g., user system interface, critical algorithms, etc.) to support design feasibility. However, the design baseline was represented on paper (PDL, simulations, flowcharts, vugraphs). These representations were vague, ambiguous and not amenable to configuration control. The degree of freedom in the design representations made it very difficult to uncover design flaws of substance, especially for complex systems with concurrent processing. Given the typical design review attitude that a design is "innocent until proven guilty", it was quite easy to assert that the design was adequate. This was primarily due to the lack of a tangible design representation from which true design flaws were unambiguously obvious. Under the Ada Process Model, design review demonstrations provide some proof of innocence and are far more efficient at identifying and resolving design flaws. The subject of the design review is not only a briefing which describes the design in human understandable terms, but also a demonstration of important aspects of the design *baseline* which verify design quality (or lack of quality).

Total Quality Management (TQM). In the Ada Process Model there are two key advantages for applying TQM. The first is the common Ada format throughout the lifecycle which permits consistent software metrics across the software development work force. Although these metrics don't all pertain to quality (many pertain to progress), they do permit a uniform communications vehicle for achieving the desired quality in an efficient manner. Secondly, the demonstrations serve to provide a common goal for the software developers. This "integrated product" is a reflection of the complete design at various phases in the life cycle for which all personnel have ownership. Rather than individually evaluating components which are owned by individuals, the demonstrations provide a mechanism for reviewing the team's product. This team ownership of the demonstrations is an important motivation for instilling a TQM attitude.

SOFTWARE QUALITY METRICS APPROACH

In essence, the approach we are taking is similar to that of [8] who proposes to measure software quality through the *absence of spoilage*. While his definitions are purposely vague (to remain technology and project independent), ours are quite explicit. The key to this metrics approach is similar to conventional cost estimation techniques such as COCOMO [3] where quantifiability and consistency of application are important. Note that software cost estimation has subjective inputs and objective outputs. Our approach will define objective inputs which may require subjective interpretation for project context.

Our primary metric for software quality will be rework as measured by changed SLOC in configured baselines. This metric will also need to be adjusted for project context to accommodate the product characteristics, the life cycle phase, etc. The software quality assessment derived from this objective collection of rework metrics will require subjective analysis in some cases. The subjectivity here is in the fact that we are trying to assess quality during development (this requires subjective analysis) using the same metrics used to assess quality following development (objective analysis). For example, the volume of rework following product delivery is an objective measure of quality, or lack of quality. The amount of rework following the first configuration baseline during development is a subjective measure. Zero rework might be interpreted as a perfect baseline (unlikely), an inadequate test program, or an unambitious first build. The following paragraphs define some of the foundations in this approach:

Software Quality Definition. *Software quality is the degree of compliance with the customer expectations of function, performance, cost and schedule.* This is an incredibly difficult concept to make

objective. The only mechanisms available for defining "customer expectations" are Software Requirements Specifications for function and performance, and an approved expenditure plan which quantifies cost and schedule goals (basically, this corresponds to the "contract"). These two mechanisms are traditionally the lowest quality products produced by a project since they are required to be agreed upon with numerous unknowns far too early in the lifecycle. The evolutionary process model and software quality metrics should provide better insight into the degree of compliance with customer expectations in the above four perspectives.

Software Change Order (SCO). A Software Change Order constitutes direction to proceed with changing a configured software component. This change may be needed to 1) rework a component with bad quality (a fix), or 2) rework a component to achieve better quality (an enhancement) or 3) accommodate a customer directed change in requirements. The difference between the first two types of rework is inherent in the *necessity* for the change. If the change is *required* for compliance with product specifications, then the rework is type 1. If the change is *desired* for cost-effectiveness, increased testability, increased usability, or other efficiency reasons (assuming the unchanged component is compliant), then the rework is type 2. In both cases, the rework should result in increased end product quality (requirements compliance per dollar), however, type 1 also indicates inadequate quality in a current baseline. In practice, differentiating between type 1 and type 2 may be quite subjective. As discussed later, most of the metrics are insensitive to the categorization, but if the differentiation is consistently applied, it can provide useful insight. Conventionally, SCOs were called Software Problem Reports (SPRs). To avoid confusion ("problem" has a negative connotation, and not all changes are necessarily problems), we have changed the terminology. The software quality metrics collection and analysis will use type 1 and type 2 SCOs in an appropriate manner. Type 3 SCOs need to be separated since they do not reflect any change in quality, they do however, redefine the customer expectations. Furthermore, Type 3 SCOs typically reflect a change which is of more global impact thereby requiring various levels of software and system engineering as well as high level regression testing. These types of SCOs will not be used in these metrics due to this wide range of variability. Rather, the data derived from type 1 and type 2 SCOs should provide a solid basis for estimating maintainability and the effort required for type 3 SCOs.

Source Lines of Code (SLOC). There has always been a controversy as to whether SLOC provides a good metric for measuring software volume (DeMarco calls this *bang*). [11] identifies some of the precautions necessary when dealing with SLOC. Upon reading open literature which discusses project productivities (SLOC/MM), it is easy to see that there is little, if any, comparability between projects within the same company no less projects from different companies. [4] identifies the pros and cons of various measures and comes to the conclusion that there is nothing better. Everyone agrees however, that whatever one uses, it must be defined objectively and consistently to be of value for comparison. How we define the absolute unit of SLOC is not as important as defining it consistently across all projects and all areas of a specific project. Therefore, the preferred way to define a SLOC is the following:

> The number of SLOC for a given set of Ada program units is defined as the output of a SLOC Counting Tool.

Enforcing this definition is simple to achieve by providing a portable tool. By accepting certain non-controversial and simple standards for program unit headers and program layout the tool can provide more valuable outputs than simply SLOC counts (e.g., static hierarchies, and complexity ratings).

Ada/COCOMO [5], [6] defines SLOC for Ada programs as: Within an Ada specification part, each carriage return counts as one SLOC. Specifications shall be coded with the following standards (rationale is provided in *italics*):

1. each parameter of a subprogram declaration be listed on a separate line (*The design of a subprogram interface is done in one place and generally the effort associated with the interface design is dependent on the number of parameters.*)

2. for custom enumeration types (e.g., system state, socket names, etc.) and record types each enumeration or field should be listed on a separate line. (*Custom types usually involve custom design and engineering, hence an increased number of SLOC.*)

3. for predefined enumeration types (e.g., keyboard keys, compass directions), enumerations should be listed on as few lines as possible without loss of readability. (*These kinds of types generally require no custom engineering.*)

4. Initialization of composite objects (e.g., records or arrays) should be listed with one component per line. (*Frequently, each of these assignments represents a custom statement, an others clause is typically used for the non-custom assignments.*)

Within Ada bodies each semi-colon counts as one SLOC. Generic instantiations count one line for each generic parameter (spec or body).

The definition above treats declarative (specification) design much more sensitively than it does executable (body) design. It also does not recognise the declarative part of a body as the same importance as a specification part. Although these and other debates can surface with respect to the "optimum" definition of a SLOC, the optimum *absolute* definition is far less important than a consistent *relative* definition.

Quality Control Board. The QCB constitutes the governing body responsible for authorizing changes to a configured baseline product (conventionally known as a configuration control board - CCB). This body is composed, at a minimum, of the development manager, customer representative, each product manager, *systems effectiveness* representative and the test manager. The QCB decides on all proposed changes to configured products and approves all SCOs. The QCB is responsible for collecting the Software Quality metrics, objectively and subjectively analyzing trends, and proposing changes to the development process, tools, products or personnel to improve future quality.

Configured Baseline. A configured baseline constitutes a set of products which are subjected to change control through a Quality Control Board (QCB). Configured baselines usually represent intermediate products which have completed design, development, and informal test and final products which have completed formal test.

METRICS DERIVATION

The remainder of this paper provides substantial detail in the definition and description of the necessary statistics to be collected, the metrics derived from these statistics and their interpretation. This section provides a simple overview of how these metrics were derived, the necessity of some of the collected statistics and their raison d'être. The following derivations are not an obvious top down progression, rather, they resulted from substantial trial and error, numerous dead end analyses, intuition and heuristics.

The fundamental hypothesis was that their was significant information content in the character of rework being performed over the project lifecycle. The obvious raw statistics to collect include number and type of software changes, SLOC damaged, and SLOC fixed. The problem was to find the right filtering techniques for the raw rework statistics which identify useful trends and to uncover objective measures which quantify product attributes both during development and as an end-product. Our original intent was to provide a quantification of the product's modularity, changeability, and maintainability. The first two are intuitively simple to define as a function of rework, the third is more subtle:

Modularity (Q_{mod}): The average extent of breakage. *This identifies the need to quantify extent of breakage* (we will use volume of SLOC damaged) and number of instances of rework (Number of SCOs). In effect we are defining modularity as a measure of breakage localization.

Changeability (Q_C): The average complexity of breakage. This identifies the need to quantify *complexity of breakage* (we will use effort required to resolve) and number of instances of rework (Number of SCOs).

Maintainability (Q_M): Theoretically the maintainability of a product is related to the productivity with which the maintenance team can operate. Productivities however, are so difficult to compare between projects that this definition was intuitively unsatisfying. If we ratio the productivity of rework to the productivity of development, we end up with a value which is independent of

productivity but yet a reflection of the complexity to change a product in relation to the complexity to develop it. This normalises out the project productivity differences and provides a relatively comparable metric. Maintainability then, will be defined as the ratio of rework productivity and development productivity. Intuitively, this value identifies a product which can be changed three times as efficiently ($Q_M = .33$) as it was developed as having a better (lower) maintainability than a product that can be changed twice as efficiently ($Q_M = .5$) as it was developed, independent of the absolute maintenance productivity realized. The statistics needed to compute these values are the total development effort, total SLOC, total rework effort and total reworked SLOC.

While the values above provide useful end-product objective measures, their intermediate values as a function of time would also provide insight during the development process into the expected end-product values. Furthermore, once we have gained some experience with maintenance of early increments, this experience should be useful for predicting the rework inherent in remaining increments.

The above brief derivation is starting to push the limits of our first goal (simplicity) and the following sections, on the surface, will appear to be somewhat complex. A few remarks about this are in order. First, there will always be a tradeoff between simplicity and real insight. Surface insight is usually attained very simply, detailed insight requires added knowledge and complexity. We have chosen a set of metrics which range from simple to moderately complex to cover the multiple perspectives needed by project management to ensure accuracy. It is not necessary to deal with these metrics as a complete set. Subsets, or different sets are also useful. Secondly, most of the analysis, mathematics and data collection inherent in these metrics should be automated so that managers need only interpret the results and understand their basis.

The above values were determined through extensive analysis, trial and error, and intuition. There are certainly other metrics derivable from rework statistics which would also provide useful insight. The following sections provide more detailed descriptions and notations for the collected statistics (Table 1), in-progress indicators (Table 2), and end-product quality metrics (Table 3). Hypothetical expectations are provided in Figure 2 for the in-progress indicators and collected statistics.

Collected Statistics

Table 1 identifies the necessary statistics which must be collected over the lifecycle to implement our proposed metrics.

Total Source Lines The $SLOC_T$ metric tracks the estimated total size of the product under development. This value may change significantly over the life of the development as early requirements unknowns are resolved and as design solutions mature. This total should also include reused software which is part of the delivered product and subject to contractor maintenance.

Configured SLOC This metric simply tracks the transition of software components from a maturing design state into a controlled configuration. For any given project, this metric will provide insight into progress and stability of the design/development team. [12] discusses some of the tradeoffs and risk management philosophy inherent in laying out an incremental build approach. For projects with reused software, there will be an early contribution to $SLOC_C$ and thus "immediate progress" and quality metrics as defined below.

Errors Real errors (type 1 SCOs) constitute an important metric from which many of the following are derived. The expectation is that the highest incidence of uncovering errors happens immediately after the turnover and decreases with time (i.e., the software matures).

Improvements The other stimulus for changing a baseline, improvements (type 2 SCOs), are also key to the assessment of quality and progress towards producing quality. The expectation for improvements is approximately inversely proportional to errors, in that as the error rate starts off high and damps out, the improvements start off low (the focus is on errors) and increase. This phenomenon is basically derived from the assumption that a fixed team is working the Test/Maintenance program and:

Statistic	Definition	Insight
Total Source Lines	$SLOC_T$ = Total Product SLOC	Total Effort
Configured Source Lines	$SLOC_C$ = Standalone Tested SLOC	Demonstrable Progress
Errors	SCO_1^o = No. of Open Type 1 SCOs SCO_1^c = No. of Closed Type 1 SCOs SCO_1 = No. of Type 1 SCOs	Test Effectiveness Test Progress Reliability
Improvements	SCO_2^o = No. of Open Type 2 SCOs SCO_2^c = No. of Closed Type 2 SCOs SCO_2 = No. of Type 2 SCOs	Value Engineering Design Progress
Open Rework	B_1 = Damaged SLOC Due to SCO_1^o B_2 =Damaged SLOC Due to SCO_2^o	Fragility Schedule Risks
Closed Rework	F_1 =SLOC Repaired after SCO_1^c F_2 =SLOC Repaired after SCO_2^c	Maturity Changeability
Total Rework	$R_1 = F_1 + B_1$ $R_2 = F_2 + B_2$	Design Quality Maintainability

Table 1: Collected Raw Data Definitions

$$Effort_{Errors} + Effort_{Improvements} = Constant$$

The actual differentiation between Type 1 and Type 2 is somewhat subjective. The metrics defined herein are not particularly sensitive to either type since they rely on the sum of the impacts from both types. However, the difference between Type 1 damage and Type 2 damage may provide useful insight as demonstrated on CCPDS-R.

Open Rework Theoretically, all rework corresponds to an increase in quality. Either the rework is necessary to remove an instance of "bad" quality (SCO_1), or to enhance a component for life cycle cost effectiveness (SCO_2). The dynamics of the rework coupled with the project schedule context must be evaluated to provide an accurate assessment of quality trends. A certain amount of rework is a necessity in a large software engineering effort. In fact, early rework is considered a sign of healthy progress in the evolutionary process model. Continuous rework, late rework, or zero rework due to the non-existence of a configured baseline are generally indicators of negative quality. Interpretation of this metric requires project context. In general however, the rework must ultimately go to zero at product delivery. In order to provide a consistent and automatable collection process, rework is defined as the number of SLOC *estimated* to change due to an SCO. The absolute accuracy of the estimates is generally unimportant and since open rework is tracked with an estimate and closed rework (see below) is tracked separately with actuals, the values continually correct themselves and remain consistent.

Closed Rework Whereas the breakage metrics estimated the damage done, the repair metrics should identify the actual damage which was fixed. Upon resolution, the corresponding breakage estimate should be updated to reflect the actual required repair that remains in the baseline. The actual SLOC *fixed* will clearly never be absolutely accurate. It will, however, be relatively accurate for assessing trends inherent in these metrics. Since *fixed* can take on several different meanings depending on what is added, deleted and changed, a consistent set of guidelines is necessary. Changed SLOC will increase R_1 without a change to $SLOC_C$. Added code will increase R_1 and $SLOC_C$, although not necessarily in the same proportion. Deleted code (not typically a problem) with no corresponding addition could reduce both R_1 and $SLOC_C$. A conventional differences tool with an appropriate preprocessor which converts properly formatted source files into a format which contains no comments and 1 SLOC per compared record would be the best method for computing changed SLOC. A simpler method (and the one used here) would be to simply estimate the magnitude of the fixed SLOC. Given the volume of changes and the need for only roughly accurate data for identifying trends, the accuracy of the raw data is relatively unimportant.

In-Progress Indicators

Table 2 defines the in-progress indicators and Figure 2 identifies relative expectations. It is difficult to define the absolute expectations for the in-progress metrics without comparable data from other projects. Relative expectations are described in the following paragraphs.

Indicator	Definition	Insight
Rework Ratio	$RR = \frac{R_1+R_2}{SLOC_C}$	Future Rework
Rework Backlog	$BB = \frac{B_1+B_2}{SLOC_C}$	Open Rework
Rework Stability	$SS = (R_1 + R_2) - (F_1 + F_2)$	Rework Trends

Table 2: In Progress Indicator Definitions

Rework Ratio The sum of the currently broken product $(B_1 + B_2)$ and the already repaired breakage $(F_1 + F_2)$ corresponds to the mass of the current product baseline which has needed rework $(R_1 + R_2)$. The rework ratio (RR) identifies the current ratio of $SLOC_C$ which is expected to undergo rework prior to maturity into an end product. The expectation for RR shown in Figure 2 is to increase to a stable value with minor discontinuities following the initial delivery of each build.

Rework Backlog The current backlog of rework is defined as the percentage of the current $SLOC_C$ which is currently in need of repair. In general, one would expect that the rework backlog should rise to some level and remain stable through the test program until it drops off to zero. Large changes from month to month should clearly be investigated.

Rework Stability The difference between total rework and closed rework provides insight into the trends of resolving issues. The important use of this metric is to ensure that the breakage rate is not outrunning the resolution rate. Figure 2 identifies an idealized case where the resolution rate does not diverge (except for short periods of time) from the breakage rate. Note also that the breakage rate somewhat tracks the $SLOC_C$ delivery rate. A diverging value of SS would indicate instability of rework activities. A stable value of SS would indicate systematic and straightforward resolution activities.

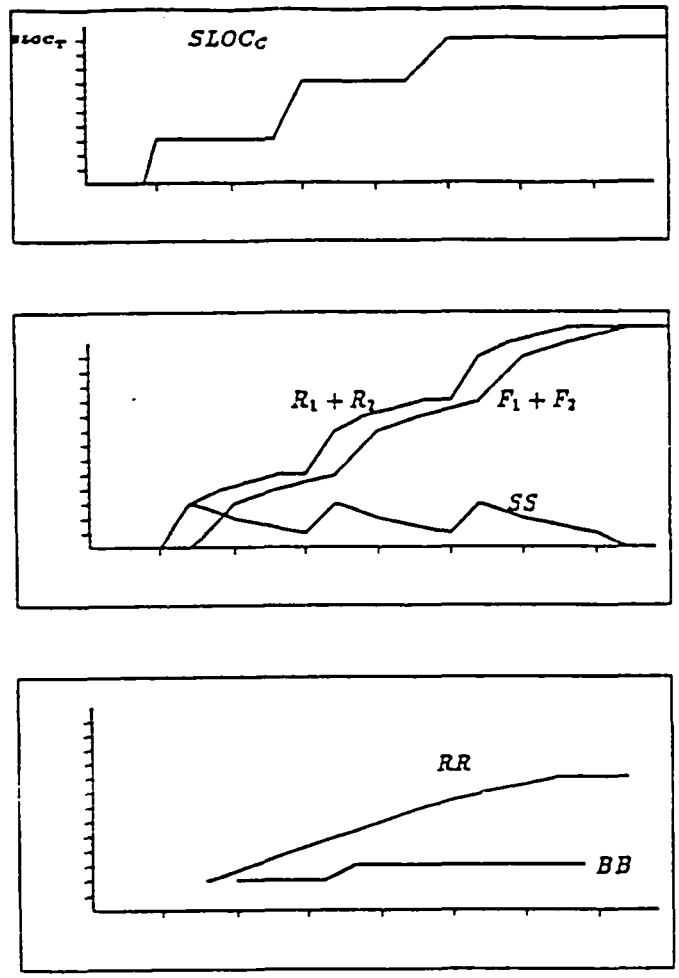


Figure 2: In-Progress Indicators Example Expectations

End-Product Quality Metrics

The end-product metrics reflect insight into the maintainability of the software products with respect to type 1 and type 2 SCOs. Type 3 SCOs are explicitly not included since they redefine the inherent target quality of the system and tend to require more global system and software engineering as well as some major re-verification of system level requirements. Since these types of changes are dealt with in extremely diverse ways by different customers and projects, they would tend to cloud the meanings and comparability of the data. However, the metrics data below should be very helpful in determining and planning the expected effort for implementing type 3 SCOs.

Rework Proportions The R_B value identifies the percentage of effort spent in rework compared to the total effort. In essence, it probably provides the best indicator of productivity. The activities included in these efforts should only include the technical requirements, software engineering, design, development, and functional test. Higher level system engineering, management, configuration control, verification testing and higher level system testing should be excluded since these

Metric	Definition	Insight
Rework Proportions	$R_E = \frac{Effort_{SCO_1}+Effort_{SCO_2}}{Effort_{Total}}$ $R_S = \frac{(R1+R2)_{Total}}{SLOC_{Total}}$	Productivity Rework Project Efficiency
Modularity	$Q_{mod} = \frac{R_1+R_2}{SCO_1+SCO_2}$	Rework Localization
Changeability	$Q_C = \frac{Effort_{SCO_1}+Effort_{SCO_2}}{SCO_1+SCO_2}$	Risk of Modification
Maintainability	$Q_M = \frac{R_E}{R_S}$	Change Productivity

Table 3: End-Product Quality Metrics Definitions

activities tend to be more a function of the company, customer or project attributes independent of quality. The goal here is to normalize the widely varying bureaucratic activities out of the metrics. R_S provides a value for comparing with similar projects, future increments, or future projects as well as other in progress analyses. Basically, it defines the proportion of the product which had to be reworked in its lifecycle. Note that the actual value could be greater than 100% .

Modularity This value identifies the average SLOC broken per SCO which reflects the inherent ability of the integrated product to localize the impact of change. To the maximum extent possible, QCBs should ensure that SCOs are written for single source changes.

Changeability This value provides some insight into the ease with which the products can be changed. While a low number of changes is generally a good indicator of a quality process, the magnitude of effort per change is sometimes even more important.

Maintainability This value identifies the relative cost of maintaining the product with respect to its development cost. For example, if $R_E = R_S$, one could conclude that the cost of modification is equivalent to the cost of development from scratch (not highly maintainable). A value of Q_M much less than 1 would tend to indicate a very maintainable product, at least with respect to development cost. Since we would intuitively expect maintenance costs of a product to be proportional to its development cost, this ratio provides a fair normalization for comparison between different projects. Since the numerator of Q_M is in terms of effort and its denominator is in terms of SLOC, it is a ratio of productivities (i.e., effort per SLOC). Some simple mathematical rearrangement will show that Q_M is equivalent to:

$$Q_M = \frac{Productivity_{Maintenance}}{Productivity_{Development}}$$

Expectations It is difficult to define the expectations for the end-product metrics without comparable data from other projects. Now that we have solid data for CCPDS-R, we can form expectations for future increments of CCPDS-R as well as other projects.

The above descriptions identify idealized trends for these metrics. Undoubtedly, real project situations will not be ideal. Their differences from ideal, however, are important for management and customer to comprehend. Furthermore, the application of these metrics on project increments as well as the project as a whole, should be useful.

APPLICATION RESULTS

Figure 3, Figure 4 and Table 4 provide the actual data to date for the CCPDS-R project. The Command Center Processing and Display System Replacement (CCPDS-R) project will provide display Information used during emergency conferences by the National Command Authorities; Chairman, Joint Chiefs of Staff; Commander in Chief North American Aerospace Command; Commander in Chief United States Space Command; Commander in Chief Strategic Air Command; and other nuclear capable Commanders in Chief. It is the missile warning element of the new Integrated Tactical Warning/Attack Assessment System developed by North American Aerospace Defense Command/Air Force Space Command.

The CCPDS-R project is being procured by Air Force Systems Command Headquarters Electronic Systems Division (ESD) at Hanscom AFB and was awarded to TRW Defense Systems Group in June 1987. TRW will build three subsystems. The first, identified as the Common Subsystem, is 30 months into development. The Common Subsystem consists of 350,000 source lines of Ada with a development schedule of 38 months. It will be a highly reliable, real-time distributed system with a sophisticated User Interface and stringent performance requirements implemented entirely in Ada. CCPDS-R Ada risks were originally a very serious concern. At the time of contract definition, Ada host and target environment, along with Ada trained personnel availability were questionable.

The data provided in this paper was collected by manually analysing 1500+ CCPDS-R SCOs maintained online and in hard copy notebooks. Most of the data defined in the previous section was available in the SCOs. Each problem description and resolution was evaluated to determine whether the SCO was type 1 or type 2 and whether the SCO was relevant to the operational product (out of the 1500 SCOs, 910 were relevant, the remainder were SCOs for initial turnovers, support tools, test software or commercial software). Furthermore, each SCO opened contained an estimate of the effort to fix and each closed SCO provided the actual (technical) effort required for the fix. The statistic which was not present, unfortunately, was the actual breakage assessment in SLOC. For each relevant SCO, the SLOC breakage estimate was based on experience with the fix, the detailed description of the resolution, the hours of analysis and the hours required for implementing the fix. While not perfectly accurate in all cases, these estimates are at least consistent relative to each other and given the large sample space, relatively accurate for the intended use. Again, it is not that important to be absolutely exact when the metrics and trends are derived from a large sample and only useful to at most 1 or 2 digits of accuracy.

CCPDS-R Common Subsystem Analysis

The following paragraphs discuss the quality metrics results for the CCPDS-R common subsystem as a whole with conclusions drawn where applicable. Figure 3 provides CCPDS-R actuals with the incremental build sequence ($SLOC_C$) overlayed for comparison.

Configured SLOC. The CCPDS-R installments of $SLOC_C$ delivered small initial builds (A0/A1 and A2) with the highest risk components. The middle build (A3), while less risky, was bulky and a substantial portion of the build was produced by (somewhat immature) automated tools. Nevertheless, it was installed in two increments (A31 and A32).

SCOs. As expected, the SCO rate is proportional to the $SLOC_C$ rate. The actuals also suggest that the state of the first two builds was higher quality at delivery than the third build. The feeling of the development managers on the project concurs with this assessment but also added that it was during the A3-A4 timeframe when substantial requirements volatility occurred in the user interface and external interface definitions. The number of open SCOs has remained fairly constant with respect to the number generated and hence indicative that the rework is being resolved in a timely fashion.

Rework Resolution. The total rework ($R_1 + R_2$) has also grown at a rate proportional to $SLOC_C$ growth but its rate of growth is decreasing. Now that the software is all configured and turnovers are complete, breakage should start damping out rapidly. The resolved rework ($F_1 + F_2$) tracked the total rework closely with little, if any divergence. The last three months indicate that the rate of resolution is exceeding the rate of breakage. This should indicate to the management team that no serious problems are lurking in the future.

Rework Ratio. The rework rate has grown from the initial builds to an apparently stable value of .15. This would imply that the initial build was more mature at delivery than the second and third builds. With over 98% of the software in $SLOC_C$, this value should be expected to be fairly stable and a

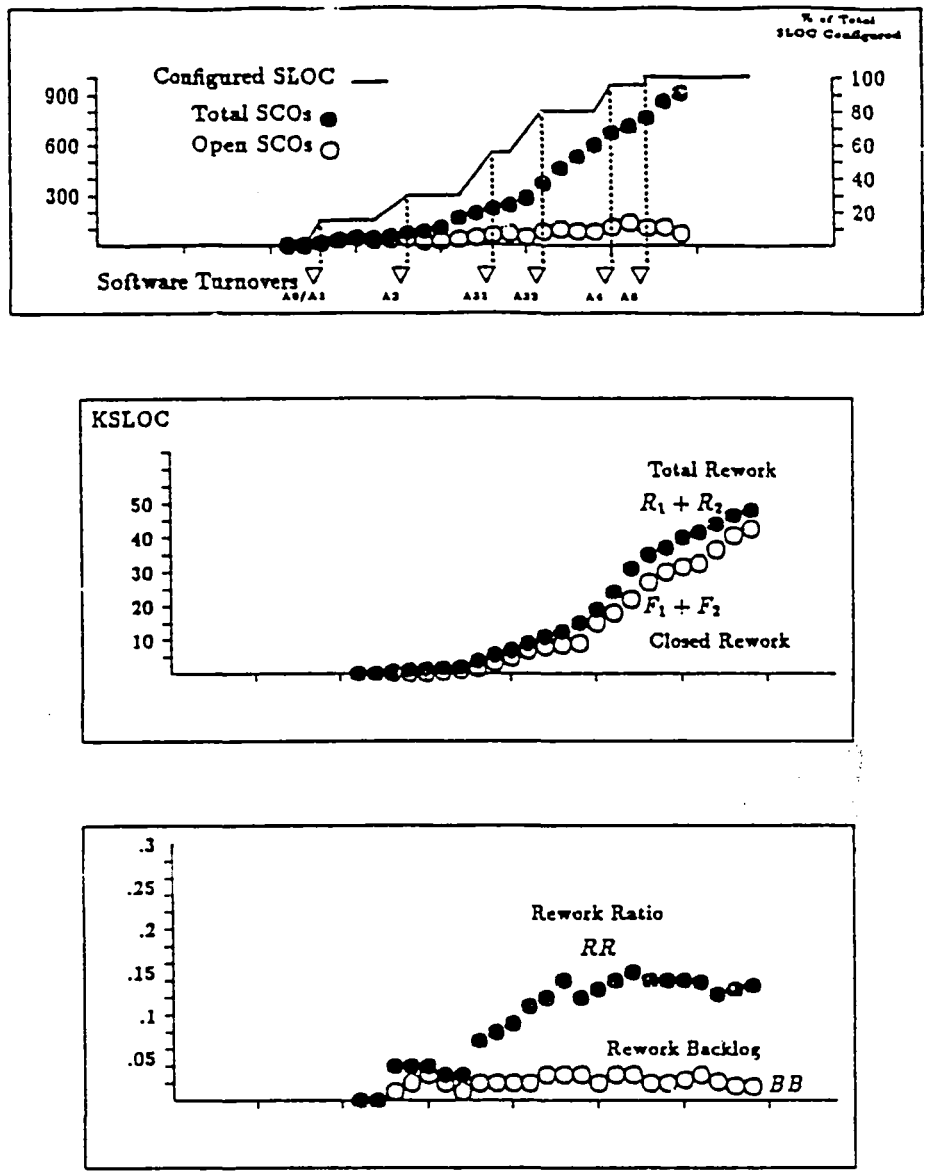


Figure 3: CCPDS-R Collected Statistics

good predictor of future rework. The amount of rework backlog in proportion to $SLOC_C$ has remained fairly constant and implies that the divergence of breakage rate and resolution rate should correct itself shortly. The situation here is that substantial increments are being added to $SLOC_C$ and an increase in breakage vs resolution is expected since the development team is likely focusing on installing baseline components rather than fixing components.

SCO Effort Distributions. Figure 4 identifies the distribution of SCOs by the effort required for resolution. This graphic also suggests that the software is generally easy to modify. A deeper analysis of the data shows that the majority of complex SCOs occurred in the more complex early builds.

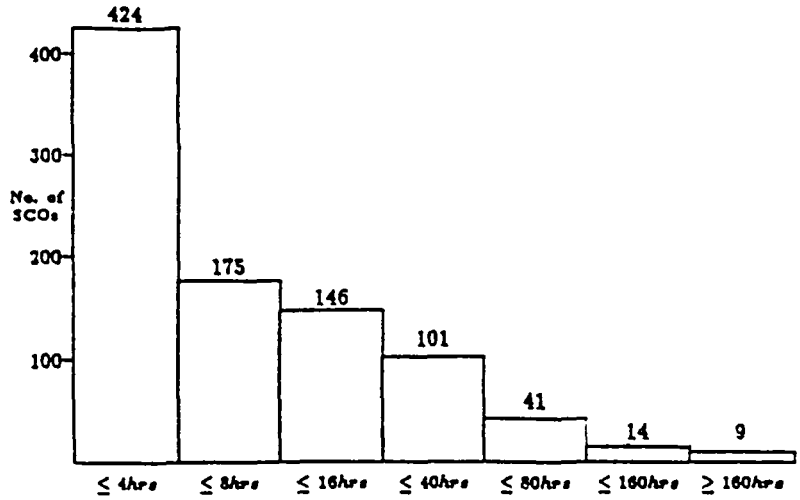


Figure 4: SCO Effort Distribution

Rework Proportions. R_E (Table 4) defines the percent of the development efforts devoted to rework. Since we only tracked the technical effort in analyzing and implementing resolutions, we have compared it to the software development effort devoted to the same, namely, the requirements, design, development and test effort. In both cases we eliminate the cost of management, facility, secretarial, configuration management, quality assurance, and other level of effort administrative activities. Note that we have included the software requirements analysis effort since, in our evolutionary approach, there is only a subtle difference between requirements and design. R_S defines the percentage of source code which has undergone rework. CCPDS-R is currently projecting a rework ratio of 14% .

Metric	Definition	CCPDS-R Value
Rework Proportions	$R_E = \frac{Effort_{SCO_1}+Effort_{SCO_2}}{Effort_{Total}}$	6.7%
	$R_S = \frac{(R1+R2)}{SLOC_C}$	13.5%
Modularity	$Q_{mod} = \frac{R_1+R_2}{SCO_1+SCO_2}$	53 $\frac{SLOC}{SCO}$
Changeability	$Q_C = \frac{Effort_{SCO_1}+Effort_{SCO_2}}{SCO_1+SCO_2}$	15.7 $\frac{Hrs}{SCO}$
Maintainability	$Q_M = \frac{R_E}{R_S}$	.49

Table 4: End-Product Quality Metrics Definitions

Modularity. This value characterizes the extent of damage expected for the average SCO. A value of 53 SLOC implies that the average SCO only affected the equivalent of one program unit. Since most of the trivial errors get caught in standalone test and demonstration activities, this value indicates the average impact for the non-trivial errors which creep into a configuration baseline. This value suggests

that the software design is flexible but with no basis for comparison, this is purely conjecture. An additional metric which would be useful in assessing modularity would be the number of files affected per change, This would provide insight into the locality of change as well as the extent. This information was not available in the CCPDS-R historical data, but it is being collected in future data.

Changeability. The average effort per SCO provides a mechanism for comparing the complexities of change. As a project average, 16 hours suggests that change is fairly simple. When change is simple, a project is likely to increase the amount of change thereby increasing the inherent quality.

Rework Improvement. Figure 5 identifies how the changeability (Q_C) evolved over the project schedule to date. While conventional experience is that changes get more expensive with time, CCPDS-R demonstrates that the cost per change improves with time. This is consistent with the goals of an evolutionary development approach [12] and the promises of a good layered architecture [13] where the early investment in the foundation components and high risk components pays off in the remainder of the life cycle with increased ease of change. The trend of this metric would indicate that the CCPDS-R software design has succeeded in providing an integrable component set with effective control of breakage. Had the trend of this metric showed growth in effort per SCO without stabilization, management may be concerned about the design quality and downstream risks in reworking an increasingly hard to change product. Note that Q_C metrics do not include the cost of downstream re-verification of higher level requirements since the broad range of these activities would corrupt the intent of the metric. Q_C has been purposely defined to reflect the technical risk of change, not the cost of reverification in a larger context or the management risk. For example, a late change of minor complexity could result in regression test by inspection or a complete reverification of numerous performance threads. This range of effort varies with the context of the change, the customer/contractor paranoia and a variety of other issues which are not reflective of the ease of change. The technical cost of change is not closed out however, until this reverification is complete since it may result in reconsideration.

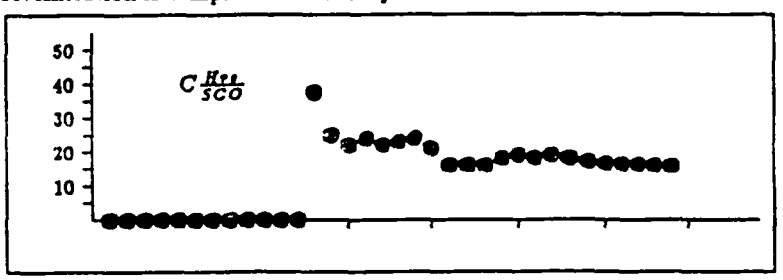


Figure 5: Rework Improvement: Changeability Evolution

Maintainability. The ratio of R_g to R_s characterizes the cost of reworking CCPDS-R components compared to developing them from scratch. This value along with the change traffic experienced during the last phase of the life cycle could be used to predict the maintenance productivity expected from the current development productivity being experienced. The overall change traffic during development should not be used to predict operational maintenance since it is overly biased by immature product changes. The FQT phase change traffic (likely a lower value than the complete development lifecycle traffic), is a more accurate measure. A value of .49 seems like a good maintainability rating, but further project data would permit a better basis for assessment.

This value requires some caveats in its usage. First, this maintenance productivity was derived from small scale maintenance actions (fixes and enhancements) as opposed to large scale upgrades where system engineering and broad redesign may be necessitated. Secondly, the data is derived from the development lifecycle, therefore, it should be treated as more of an upper bound in planning the expectations during the maintenance phase of a product where the existence of defects should be less than that experienced during development. The personnel performing the maintenance actions however, were *knowledgeable developers* which may bias the maintainability compared to the expertise of the maintenance team. The message here, is that this data, like any productivity data, must be used carefully by people cognisant with its derivation to ensure proper usage.

Functional CSCI Analysis. A complete lower level analysis was performed to analyse the various contributions to the values in Table 4 by the individual CSCIs. While the evaluation of this lower level data will not be discussed here in detail, they did uncover some interesting phenomena which have since been incorporated into the plans of future subsystems. There were significant differences in the various CSCI level values which provided insight into various levels of quality and the need for perturbations to future plans. The Q_M varied from .12 to .85 across 6 CSCIs. For example, relatively low values were observed for algorithm (.12) and display (.27) software where ease of change was a clear design goal. Higher values were observed for the external communications software (.51) and system services software (.85) where changes in an external message set for example, could result in broader system impacts. The range of values clearly identifies the *relative* difference in risk associated with changing various aspects of the design. The absolute risk associated with these changes is difficult to assess without further data from other similar projects.

Global Summary. In general, the CCPDS-R program appears to be converging towards a very high quality product with high probability. This assessment is implied from the visible stability in the quality metrics. The fact that these metrics are stable generally implies that the remaining efforts are predictable. If the predictions do not extrapolate to better than required performance, action can be taken. The key to optimising the value of these metrics is to achieve stabilization as early as possible so that if predicted performance does not match expectations, management can instigate improvement actions as early in the life cycle as possible. Some characteristics of CCPDS-R which are important to keep in mind when interpreting the above metrics include:

1. Many changes incurred by the project were really type 3 (true requirements change). However, since most of these were small it was easier to incorporate them rather than go through the formal ECP process. In retrospect, the sum of all these little changes was quite substantial.

2. These metrics are derived from the development phase, comparison with other project's maintenance phase metrics is misleading. The metrics available in the final 3 months prior to delivery (as opposed to the lifecycle averages presented here) however, should be fairly comparable.

Operational Concept. The concept of operations for the software quality metrics program is to provide insight for the purposes of managing product development with minimum interference to the development team. This will be accomplished by integrating the standards for metrics collection into the tools and QCB procedures. The responsibilities of this initiative are allocated as follows:

Software Developers: Follow the core Ada Design/Development Standards

Software Development Managers: Follow the evolutionary process model, adhere to core software quality metrics policy, coordinate with project systems effectiveness any project unique policies, interpret systems effectiveness SQM analysis and be accountable for issues and resolutions.

Corporate Systems Effectiveness: Define the SQM policy/tools/procedures, evaluate project implementations, improve the policies/tools/procedures and ensure consistent usage across different projects. This is the same function proposed by [8] as the standards group.

Project Software Engineering: Flowdown the SQM policy/tools/procedures into a project implementation, implement project QCB, SQM collection, SQM analysis, SQM reporting, evaluate project implementations, and propose candidate improvements to the policies/tools/procedures. Note that we are putting this function in the hands of knowledgeable project personnel (as opposed to conventional independent QA personnel) since the administrators of these metrics should be motivated for effective use through ownership in both the process and the products.

We would foresee SQM metrics reporting on a monthly or quarterly basis depending on project phase, size, risks, etc. Furthermore, the entire SQM initiative should be relatively dynamic during its infancy as real project applications determine what is most useful and feedback is incorporated.

SUMMARY

By itself, CCPDS-R is perhaps a bad example for testing these metrics. In general, the project has performed as planned and has a high probability of delivering a quality product. It would be useful to examine a less successful project to illustrate the tendencies which every project manager should be looking for as indicators of trouble ahead.

Furthermore, none of these metrics by themselves, provides enough data to make an assessment of a project's quality. They must be examined as a group in conjunction with other conventional measures to arrive at an accurate assessment. They also do not represent the only set of useful metrics possible from the collected statistic on SCOs and rework. There are many other ways to examine this data and present it for trend analysis. With further automation, these other views would be simple to produce.

Although not fully implemented on a large project to date, subsets of the metrics presented herein have proven useful in the long term planning and development process improvement on CCPDS-R. TRW is currently in the process of expanding these concepts into a uniform practice across its Ada software development projects supported by automated tools. With the broad acceptance of Ada and evolutionary development techniques, this approach has the potential of providing a uniform technique for quality metrics collection, reporting and history. This data is paramount to the implementation of a consistent TQM approach to software development for enhanced software product quality and more efficient software production. The following activities still need to be performed to provide a complete initiative:

1. Enhance the standard SCO form with definitions, standards and procedures for usage.
2. Develop a portable SLOC Counting Tool (the current CCPDS-R Metrics Tool would satisfy this with minor modifications).
3. Identify Ada standards (which would be mandatory across all Ada projects) necessary to guarantee consistent metrics collection across projects and within projects. This primarily involves standards for program unit headers and program layout which are not controversial.
4. Develop an SCO data base management system with supporting tools for automated collection, analysis and reporting in the formats defined above and other, as yet undiscovered, useful formats.
5. Define QCB procedures, guidelines for metrics analysis and candidate reporting formats.
6. Incorporate this initiative into corporate policy.

As a conclusion, we should evaluate the approach presented herein with our original goals:

1. **Simplicity.** The number of statistics to be maintained in an SCO database to implement this approach is 5 (type, estimate of damage in hours and SLOC, actual hours and actual SLOC to resolve) along with the other required parameters of an SCO. Furthermore, metrics for $SLOC_C$ and $SLOC_T$ need to be accurately maintained. If automated in an online DBMS, the remaining metrics could be computed and plotted from various perspectives (e.g., by build, by CSCI) in a straightforward manner. Depending on the extent of discipline already inherent in a project's CCB and development metrics, the above effort could be viewed as very simple (as in the case of CCPDS-R) to complex (undisciplined, management by conjecture projects).
2. **Ease of Use.** The metrics described herein were easy to use by CCPDS-R project personnel and managers familiar with the project context. Furthermore, they provide an objective basis for discussing current trends and future plans with outside authorities and customers. Most trends are obvious and easily explained. Some trends require further analysis to understand the underlying subtleties. End-product metrics provide simple to understand indicators of different software quality aspects for the purposes of comparison and future planning as well as assessment of process improvement.

3. **Probability of Misuse** There are enough perspectives that provide somewhat redundant views so that misuse should be minimized. Without further experience, however, it is not clear that contractor and customer will always interpret them correctly. Although correct interpretation could never be guaranteed, it would be beneficial to obtain more experience to evaluate where misinterpretation is most likely.

BIOGRAPHY

Walker Royce is currently the Principal Investigator for an Independent Research and Development Project directed at expanding distributed Ada architecture technologies proven on CCPDS-R. He received his BA in Physics at the University of California, Berkeley in 1977, MS in Computer Information and Control Engineering at the University of Michigan in 1978, and has 3 further years of post-graduate study in Computer Science at UCLA. Mr. Royce has been at TRW for 12 years, dedicating the last six years to advancing Ada technologies in both research and practice. He served as the Software Chief Engineer responsible for the software process, the foundation Ada components and the software architecture on the CCPDS-R Project from 1987-1990. From 1984-1987, he was the Principal Investigator of SEDD's Ada Applicability for C^3 Systems Independent Research and Development Project. This IR& D project resulted in the foundations for Ada COCOMO, the Ada Process Model and the Network Architecture Services Software, technologies which earned TRW's Chairman's Award for Innovation and have since been transitioned from research into practice on real projects.

REFERENCES

[1] Andres, D. H., "Software Project Management Using Effective Process Metrics: The CCPDS-R Experience" *AFCEA Military/Government Computing Conference Proceedings*, Washington D.C., January 1990.

[2] Boehm, B. W., J. R. Brown, H. Kaspar, M. Lipow, G. MacLeod, and M. J. Merrit, "Characteristics of Software Quality," *TRW Series of Software Technology*, Volume 1, TRW and North Holland Publishing, 1978.

[3] Boehm, B. W., *Software Engineering Economics*, Prentice Hall, 1983.

[4] Boehm, B. W., "Improving Software Productivity", *Computer*, September 1987.

[5] Boehm, B. W., Royce, W. E., "TRW IOC Ada COCOMO: Definition and Refinements", *Proceedings of the 4th COCOMO Users Group*, Pittsburgh, November 1988.

[6] Boehm, B. W., Royce, W. E., "COCOMO Ada et le Modele de Developpement Ada", *Genie Logiciel*, December 1989, pp. 36-53.

[7] Boehm, B. W., "The Spiral Model of Software Development and Enhancement", *Proceedings Of The International Workshop On The Software Process And Software Environments*, Coto de Caza, CA, March 1985.

[8] DeMarco, T., *Controlling Software Projects*, Yourden Press, 1982.

[9] Grady, R. B., and Caswell, D. L., *Software Metrics: Establishing a Company Wide Program*, Prentice Hall, 1986.

[10] Humphrey, W., *Managing the Software Process*, SEI Series in Software Engineering, Addison Wesley, 1989.

[11] Jones, C., *Programming Productivity: Issues for the Eighties*, IEEE Computer Society Press, 1981.

[12] Royce, W. E., "TRW's Ada Process Model For Incremental Development of Large Software Systems", *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, March 26-30 1990.

[13] Royce, W. E., "Reliable, Reusable Ada Components For Constructing Large, Distributed Multi-Task Networks: Network Architecture Services (NAS)", *TRI-Ada Proceedings*, Pittsburgh, October 1989.

[14] Shooman, M. L., *Software Engineering*, McGraw Hill, 1983.

[15] Springman, M. C., "Incremental Software Test Methodology For A Major Government Ada Project ", *TRI-Ada Proceedings*, Pittsburgh, October 1989.

[16] Springman, M. C., "Developing Maintainable & Reliable Ada Software, A Large Military Application's Experience", *Ada Europe Proceedings*, Dublin, Ireland, June 1990.

VIEWGRAPH MATERIALS

FOR THE

W. ROYCE PRESENTATION

6269-0

Pragmatic Quality Metrics For Evolutionary Software Development Models

W. E. Royce

November 1990

PRECEDING PAGE BLANK NOT FILMED

TRW	Large Software Project Issues	TRW

Primary Contributors To Software Diseconomy of Scale (Boehm):

- Rework
- Interpersonal Communications
- Requirements Volatility

Ada COCOMO (Boehm/Royce) Speculates That Economy of Scale Is Possible

- Use an Evolutionary Development Approach
- Use Ada as a Lifecycle Language

TRW	Evolutionary Development Objectives	TRW

Optimize Rework

- Minimize Ineffective Rework
  - Do Hard Parts First
  - Compartmentalized Breakage
- Maximize Rework Efficiency
  - Fix it Early
  - *Design for Change*

Minimize Interpersonal Communications

- Small Expert Design Team
- Layered Architecture
- Self Documenting Lifecycle Language (Ada)

Optimize Requirements Volatility

- Stablize Necessary Primitives Early
- Change Requirements As Product Matures
- *Design for Change*

SQM Focus

Most Important Software Quality is that it is SOFT

- **Modularity:**
  - Breakage Extent When Changed
- **Changeability:**
  - Complexity of Effort to Analyze/Implement Change
- **Maintainability:**
  - Productivity of Change

Assumptions:

- Evolutionary Process Model
- Consistent SLOC Counting
- Configuration Control Board For Change Assessment

SQM Definition

Quality: *Degree of compliance with customer expectations of function, performance, cost and schedule*

Quality Metrics Derived from Measurement of Rework

- Type 1 Rework: Fix Bad Quality Instance
- Type 2 Rework: Improve Quality
- Type 3 Rework: Requirements Change

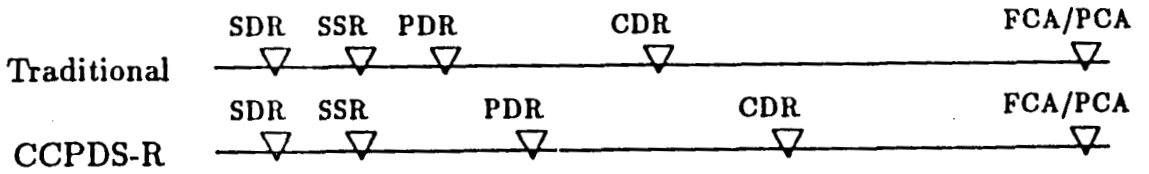
All Rework Corresponds to Quality Increase

Quality Metrics Approach

- Collect Statistics on Rework over Project Lifecycle
- Quantify Meaningful Metrics
- Plot Processed Statistics Over Time
  - Quality Progress Trends

TRW	CCPDS-R CDR STATUS	TRW

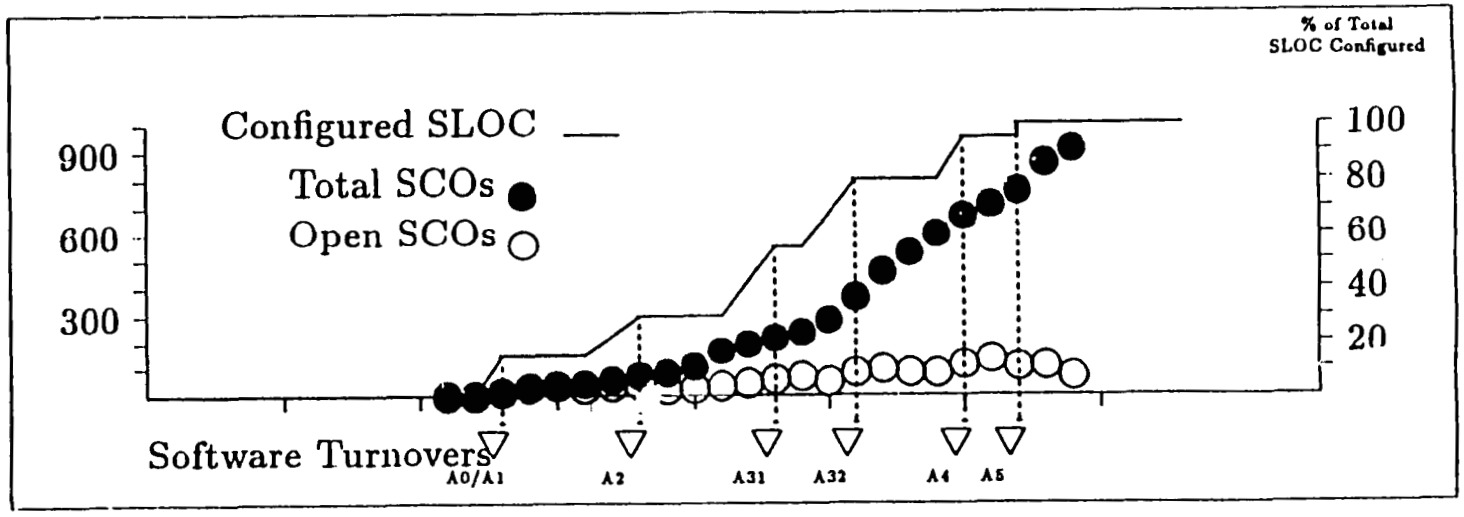
- Evolutionary Approach Permits Tangible Insight into End-product



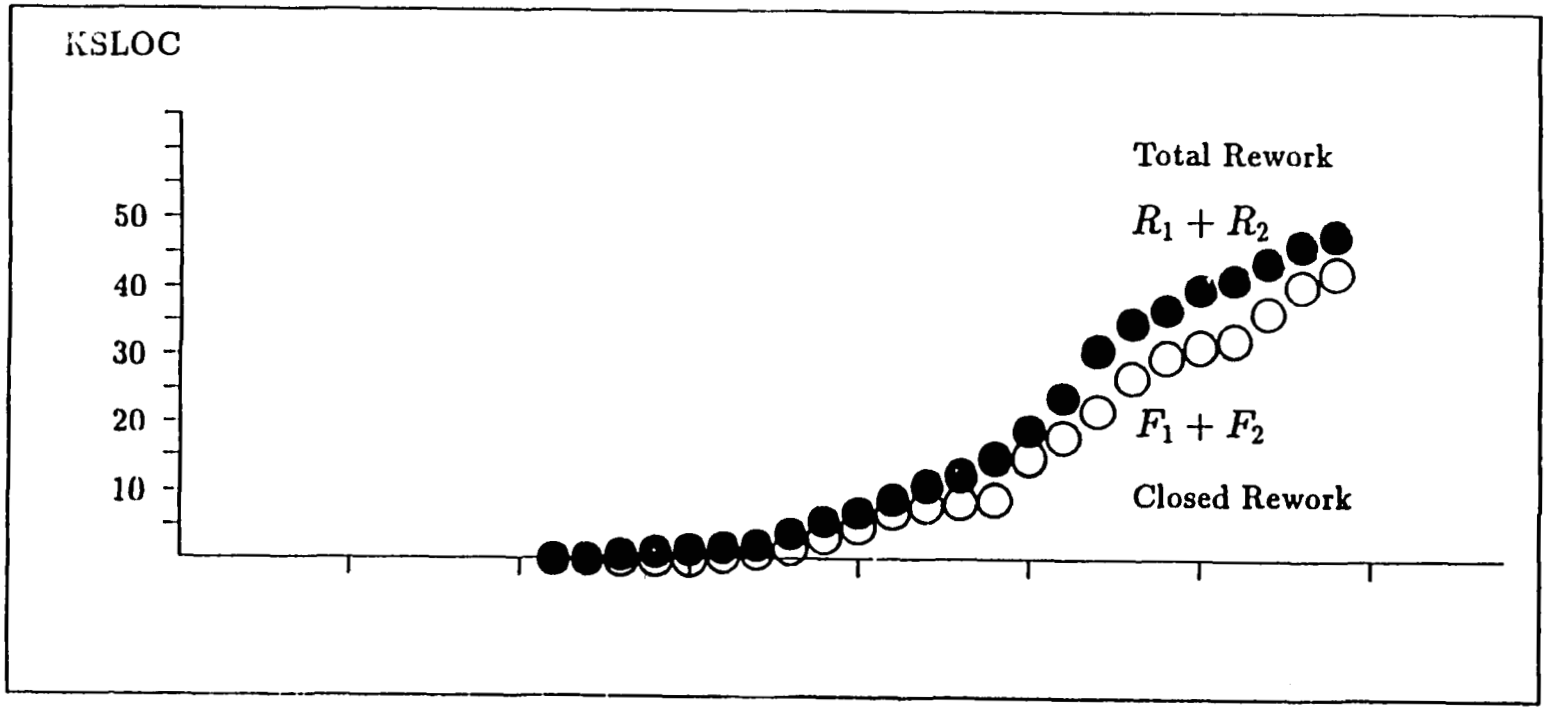
	Traditional Approach	CCPDS-R Approach
Software Design	Complete	Complete
Code development	≈10%	94%
Software Integration	Negligible	74%
Formal Test	0%	12%
Performance Assessment	Modeling	80% of Operational S/W Demonstrated

This New Approach Enables Early Software Quality Assessment

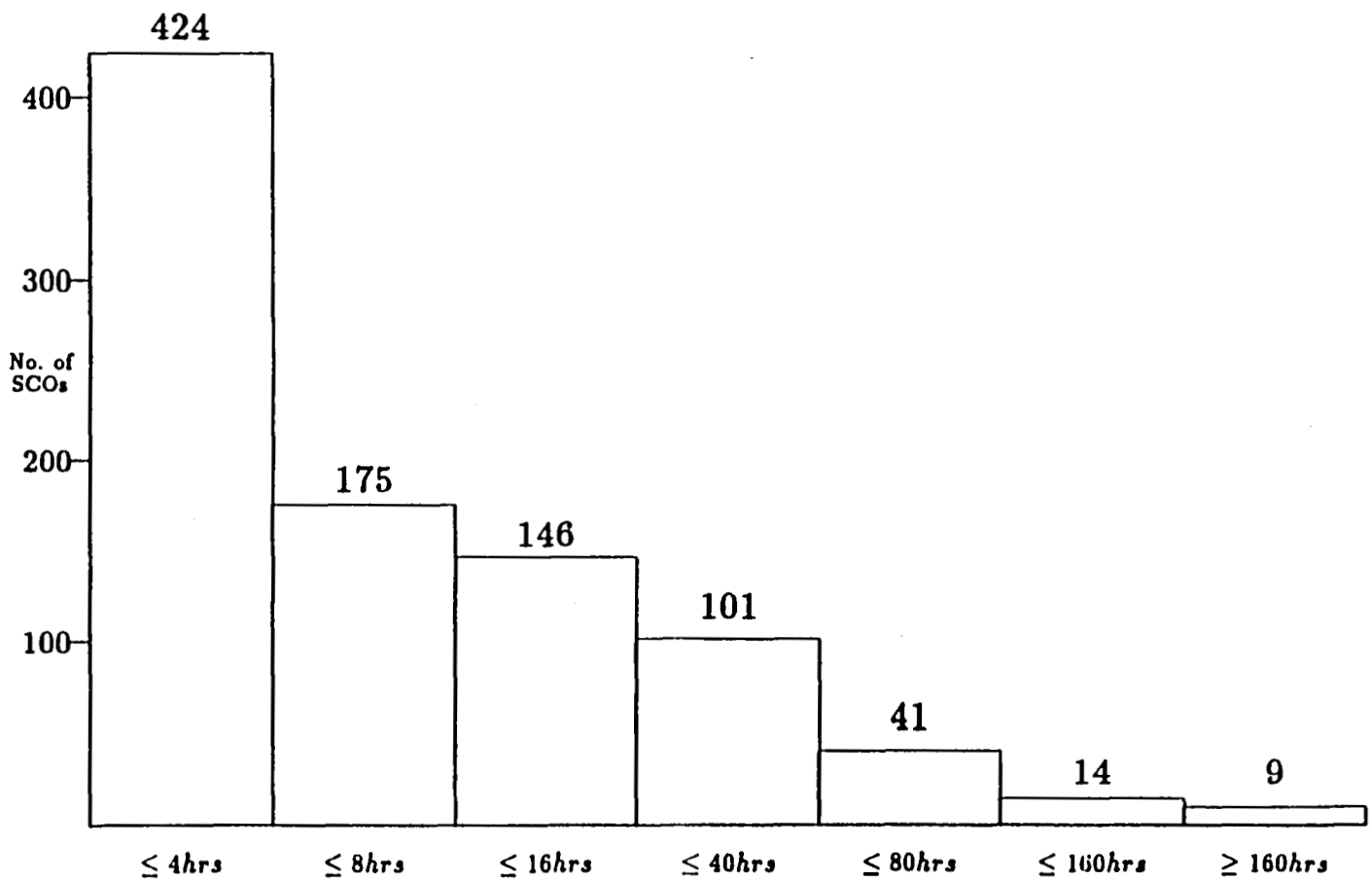
TRW	CCPDS-R Experience	TRW



TRW	CCPDS-R Experience	TRW



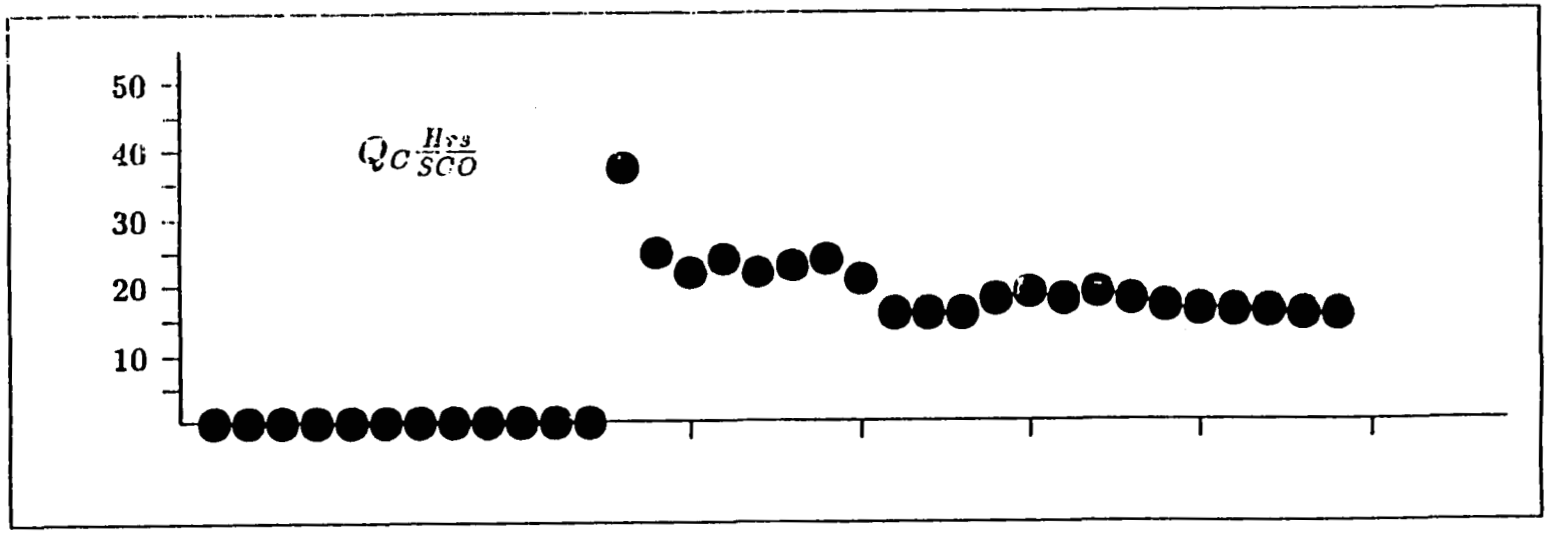
TRW	CCPDS-R SCO Experience	TRW



TRW	CCPDS-R Quality Metrics Actuals	TRW

Metric	Definition	CCPDS-R Value
Rework Proportions	Rework: R_E =% of Effort Rework: R_S =% of Product	6.7% 13.5%
Modularity	Q_{mod} =Average Breakage per Change	53 $\frac{SLOC}{SCO}$
Changeability	Q_C =Average Effort per Change	15.7 $\frac{Hrs}{SCO}$
Maintainability	Q_M =Normalized Rework Productivity	.49

TRW	CCPDS-R Changeability Evolution	TRW



W. Royce
TRW

TRW	Conclusions	TRW

Important Needs For Successful Use:

- Consistency of Application
- Automated Tools
- Management **And** Practitioner Acceptance

Advantages:

- Quantitative Data For Decision Making
- Quantitative Data For Subjective Requirements Compliance
  - Maintainability, Modularity, Adaptability, etc.
- Historical Data for Better Future Planning

Disadvantages:

- No Existing Multi-project Historical Database
  - Only CCPDS-R Data Exists Now
- No Existing Project Independent Toolsuite

Quality Metrics *Can* Be Used Effectively