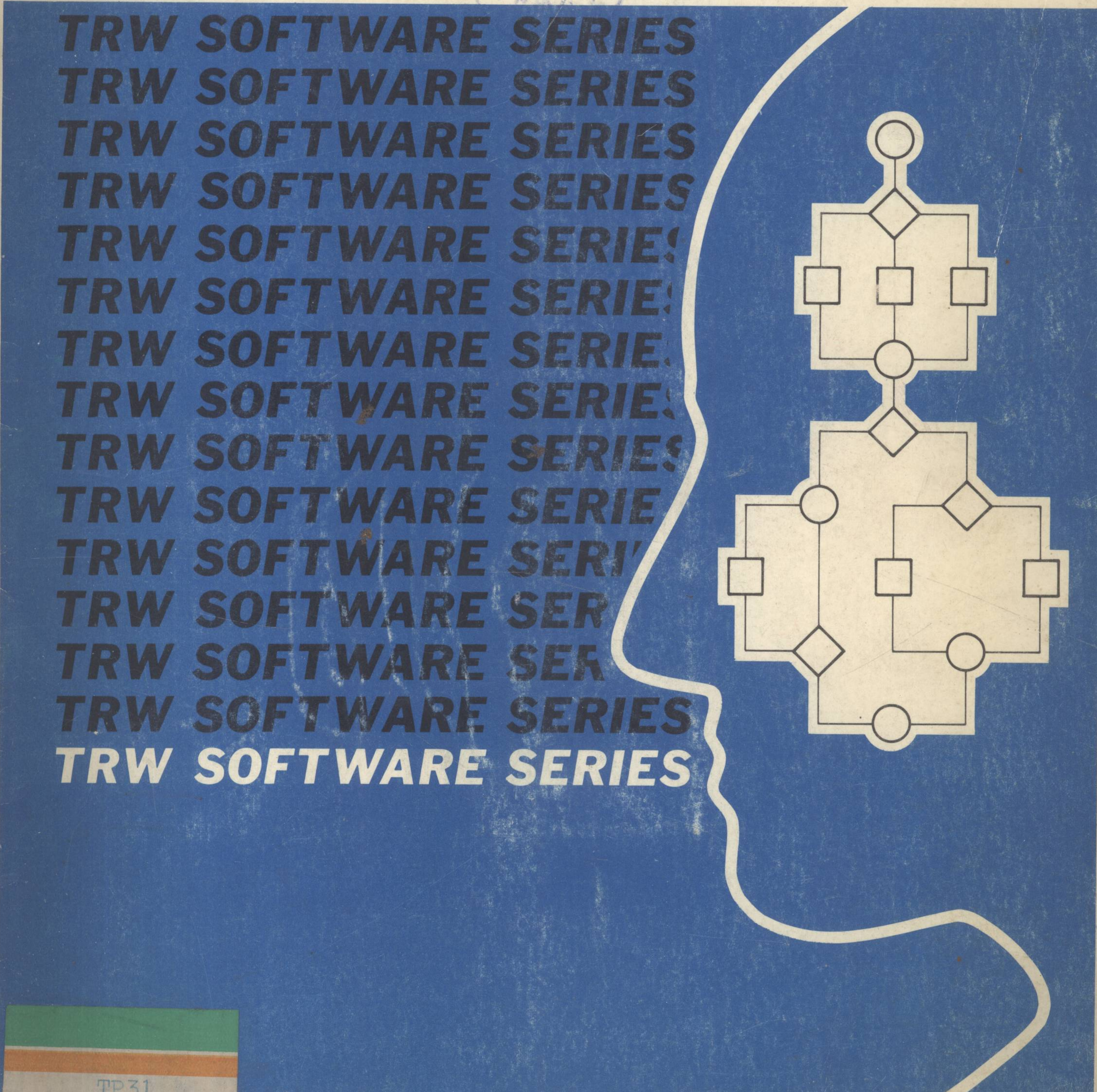
TRW-SS-73-09

CHARACTERISTICS OF SOFTWARE QUALITY

DECEMBER 1973



TRW-SS-73-09

CHARACTERISTICS OF SOFTWARE QUALITY

Prepared By

B.W. Boehm
J.R. Brown
H. Kaspar
M. Lipow
G.J. MacLeod
M.J. Merritt

28 December 1973

TRW SYSTEMS GROUP
SYSTEMS ENGINEERING AND INTEGRATION DIVISION
ONE SPACE PARK, REDONDO BEACH, CALIFORNIA 90278

ABSTRACT

This is the report of a small study by TRW for the National Bureau of Standards' Institute for Computer Sciences and Technology. The objectives of this study were to identify a set of characteristics of quality software and, for each characteristic, to define a metric such that:

1) Given an arbitrary program, the metric provides a quantitative measure of the degree to which the program has the associated characteristic, and

2) Overall software quality can be defined as some function of the values of the metrics.

Although "software" can have many components such as functional specifications, test plans, and data collection guidelines, this study concentrates on metrics which are applied to FORTRAN source programs (which may include extensive use of comment cards). However, many of the metrics in the report can be adapted straightforwardly to other software components.

The major results of the study are the following:

1) An assessment of the limitations of purely quantitative measures of software quality.

2) A definitive hierarchy of characteristics of software quality, the "Characteristics Tree".

3) An extensive list of anomaly-detecting metrics, thoroughly classified with respect to the characteristics of software quality and evaluated with respect to a set of "Characteristics of quality metrics."

4) A detailed algorithm and guidelines for using metrics to perform assessment of program header comments.

5) A discussion and cost-benefit analysis of the potential impact of using anomaly-detecting metrics during the software process.

6) An annotated bibliography of relevant literature.

The study's most important contribution though, has been to provide, for the first time, a clear, well-defined framework for assessing the often slippery issues associated with software quality, via the consistent and mutually supportive sets of definitions, distinctions, guidelines, and experience summaries cited above. This framework is certainly not complete, but it has been brought to a point sufficient to support the evaluation of the relative cost-effectiveness of prospective code-analysis tools presented in this report, and to serve as a viable basis for future refinements and extensions.

This is a reprint of TRW document 25201-6001-RU-00
prepared for the National Bureau of Standards under
contract number 3-36012
Printed in U.S.A.

Table A-2. Two Subroutines from IBM Scientific Subroutine Package

Subroutine PNORM

```
C
C     ..................................................................
C
C        SUBROUTINE PNORM
C
C        PURPOSE
C           NORMALIZE COEFFICIENT VECTOR OF A POLYNOMIAL
C
C        USAGE
C           CALL PNORM(X,IDIMX,EPS)
C
C        DESCRIPTION OF PARAMETERS
C           X      - VECTOR OF ORIGINAL COEFFICIENTS, ORDERED FROM
C                    SMALLEST TO LARGEST POWER. IT REMAINS UNCHANGED
C           IDIMX  - DIMENSION OF X. IT IS REPLACED BY FINAL DIMENSION
C           EPS    - TOLERANCE BELOW WHICH COEFFICIENT IS ELIMINATED
C
C        REMARKS
C           IF ALL COEFFICIENTS ARE LESS THAN EPS, RESULT IS A ZERO
C           POLYNOMIAL WITH IDIMX=0 BUT VECTOR X REMAINS INTACT
C
C        SUBROUTINES AND FUNCTION SUBPROGRAMS REQUIRED
C           NONE
C
C        METHOD
C           DIMENSION OF VECTOR X IS REDUCED BY ONE FOR EACH TRAILING
C           COEFFICIENT WITH AN ABSOLUTE VALUE LESS THAN OR EQUAL TO EPS
C
C     ..................................................................
C
      SUBROUTINE PNORM(X,IDIMX,EPS)
      DIMENSION X(1)
C
    1 IF(IDIMX) 4,4,2
    2 IF(ABS(X(IDIMX))-EPS) 3,3,4
    3 IDIMX=IDIMX-1
      GO TO 1
    4 RETURN
      END
```

Subroutine PDIV

```
C
C     ..................................................................
C
C        SUBROUTINE PDIV
C
C        PURPOSE
C           DIVIDE ONE POLYNOMIAL BY ANOTHER
C
C        USAGE
C           CALL PDIV(P,IDIMP,X,IDIMX,Y,IDIMY,TOL,IER)
C
C        DESCRIPTION OF PARAMETERS
C           P     - RESULTANT VECTOR OF INTEGRAL PART
C           IDIMP - DIMENSION OF P
C           X     - VECTOR OF COEFFICIENTS FOR DIVIDEND POLYNOMIAL,
C                   ORDERED FROM SMALLEST TO LARGEST POWER. IT IS
C                   REPLACED BY REMAINDER AFTER DIVISION.
C           IDIMX - DIMENSION OF X
C           Y     - VECTOR OF COEFFICIENTS FOR DIVISOR POLYNOMIAL,
C                   ORDERED FROM SMALLEST TO LARGEST POWER
C           IDIMY - DIMENSION OF Y
C           TOL   - TOLERANCE VALUE BELOW WHICH COEFFICIENTS ARE
C                   ELIMINATED DURING NORMALIZATION
C           IER   - ERROR CODE. 0 IS NORMAL, 1 IS FOR ZERO DIVISOR
C
C        REMARKS
C           THE REMAINDER R REPLACES X.
C           THE DIVISOR Y REMAINS UNCHANGED.
C           IF DIMENSION OF Y EXCEEDS DIMENSION OF X, IDIMP IS SET TO
C           ZERO AND CALCULATION IS BYPASSED
C
C        SUBROUTINES AND FUNCTION SUBPROGRAMS REQUIRED
C           PNORM
C
C        METHOD
C           POLYNOMIAL X IS DIVIDED BY POLYNOMIAL Y GIVING INTEGER PART
C           P AND REMAINDER R SUCH THAT X = P*Y + R.
C           DIVISOR Y AND REMAINDER  VECTOR GET NORMALIZED.
C
C     ..................................................................
C
      SUBROUTINE PDIV(P,IDIMP,X,IDIMX,Y,IDIMY,TOL,IER)
      DIMENSION P(1),X(1),Y(1)
C
      CALL PNORM (Y,IDIMY,TOL)
      IF(IDIMY) 50,50,10
   10 IDIMP=IDIMX-IDIMY+1
      IF(IDIMP) 20,30,60
C
C     DEGREE OF DIVISOR WAS GREATER THAN DEGREE OF DIVIDEND
C
   20 IDIMP=0
   30 IER=0
   40 RETURN

      Y IS ZERO POLYNOMIAL
C
   50 IER=1
      GO TO 40
C
C     START REDUCTION
C
   60 IDIMX=IDIMY-1
      I=IDIMP
   70 II=I+IDIMX
      P(I)=X(II)/Y(IDIMY)
C
C     SUBTRACT MULTIPLE OF DIVISOR
C
      DO 80 K=1,IDIMX
      J=K-1+I
      X(J)=X(J)-P(I)*Y(K)
   80 CONTINUE
      I=I-1
      IF(I) 90,90,70
C
C     NORMALIZE REMAINDER POLYNOMIAL
C
   90 CALL PNORM(X,IDIMX,TOL)
      GO TO 30
      END
```

Table A-1. An Example Listing from IBM Scientific Subroutines Package

```
C
C     ..................................................................
C
C        SUBROUTINE KOLMO
C
C        PURPOSE
C           TESTS THE DIFFERENCE BETWEEN EMPIRICAL AND THEORETICAL
C           DISTRIBUTIONS  USING THE KOLMOGOROV-SMIRNOV TEST
C
C        USAGE
C           CALL KOLMO(X,N,Z,PROB,IFCOD,U,S,IER)
C
C        DESCRIPTION OF PARAMETERS
C           X     - INPUT VECTOR OF N INDEPENDENT OBSERVATIONS.  ON
C                   RETURN FROM KOLMO, X HAS BEEN SORTED INTO A
C                   MONOTONIC NON-DECREASING SEQUENCE.
C           N     - NUMBER OF OBSERVATIONS IN X
C           Z     - OUTPUT VARIABLE CONTAINING THE GREATEST VALUE WITH
C                   RESPECT TO X OF  SQRT(N)*ABS(FN(X)-F(X)) WHERE
C                   F(X) IS A  THEORETICAL DISTRIBUTION FUNCTION AND
C                   FN(X) AN EMPIRICAL DISTRIBUTION FUNCTION.
C           PROB  - OUTPUT VARIABLE CONTAINING THE PROBABILITY OF
C                   THE STATISTIC BEING GREATER THAN OR EQUAL TO Z IF
C                   THE HYPOTHESIS THAT X IS FROM THE DENSITY UNDER
C                   CONSIDERATION IS TRUE.  E.G., PROB = 0.05 IMPLIES
C                   THAT ONE CAN REJECT THE NULL HYPOTHESIS THAT THE SET
C                   X IS FROM THE DENSITY UNDER CONSIDERATION WITH 5 PER
C                   CENT PROBABILITY OF BEING INCORRECT.  PROB = 1. -
C                   SMIRN(Z).
C           IFCOD- A CODE DENOTING THE PARTICULAR THEORETICAL
C                   PROBABILITY DISTRIBUTION FUNCTION BEING CONSIDERED.
C                   = 1---F(X) IS THE NORMAL PDF.
C                   = 2---F(X) IS THE EXPONENTIAL PDF.
C                   = 3---F(X) IS THE CAUCHY PDF.
C                   = 4---F(X) IS THE UNIFORM PDF.
C                   = 5---F(X) IS USER SUPPLIED.
C           U     - WHEN IFCOD IS 1 OR 2, U IS THE MEAN OF THE DENSITY
C                   GIVEN ABOVE.
C                   WHEN IFCOD IS 3, U IS THE MEDIAN OF THE CAUCHY
C                   DENSITY.
C                   WHEN IFCOD IS 4, U IS THE LEFT ENDPOINT OF THE
C                   UNIFORM DENSITY.
C                   WHEN IFCOD IS 5, U IS USER SPECIFIED.
C           S     - WHEN IFCOD IS 1 OR 2, S IS THE STANDARD DEVIATION OF
C                   DENSITY GIVEN ABOVE, AND SHOULD BE POSITIVE.
C                   WHEN IFCOD IS 3, U - S SPECIFIES THE FIRST QUARTILE
C                   OF THE CAUCHY DENSITY.  S SHOULD BE NON-ZERO.
C                   IF IFCOD IS 4, S IS THE RIGHT ENDPOINT OF THE UNIFORM
C                   DENSITY.  S SHOULD BE GREATER THAN U.
C                   IF IFCOD IS 5, S IS USER SPECIFIED.
C           IER   - ERROR INDICATOR WHICH IS NON-ZERO IF S VIOLATES ABOVE
C                   CONVENTIONS.  ON RETURN NO TEST HAS BEEN MADE, AND X
C                   AND Y HAVE BEEN SORTED INTO MONOTONIC NON-DECREASING
C                   SEQUENCES.  IER IS SET TO ZERO ON ENTRY TO KOLMO.
C                   IER IS CURRENTLY SET TO ONE IF THE USER-SUPPLIED PDF
C                   IS REQUESTED FOR TESTING.  THIS SHOULD BE CHANGED
C                   (SEE REMARKS) WHEN SOME PDF IS SUPPLIED BY THE USER.
C
C        REMARKS
C           N SHOULD BE GREATER THAN OR EQUAL TO 100.  (SEE THE
C           MATHEMATICAL DESCRIPTION GIVEN FOR THE PROGRAM SMIRN,
C           CONCERNING ASYMPTOTIC FORMULAE)  ALSO, PROBABILITY LEVELS
C           DETERMINED BY THIS PROGRAM WILL NOT BE CORRECT IF THE
C           SAME SAMPLES ARE USED TO ESTIMATE PARAMETERS FOR THE
C           CONTINUOUS DISTRIBUTIONS WHICH ARE USED IN THIS TEST.
C           (SEE THE MATHEMATICAL DESCRIPTION FOR THIS PROGRAM)
C           F(X) SHOULD BE A CONTINUOUS FUNCTION.
C           ANY USER SUPPLIED CUMULATIVE PROBABILITY DISTRIBUTION
C           FUNCTION SHOULD BE CODED BEGINNING WITH STATEMENT 26 BELOW,
C           AND SHOULD RETURN TO STATEMENT 27.
C
C           DOUBLE PRECISION USAGE---IT IS DOUBTFUL THAT THE USER WILL
C           WISH TO PERFORM THIS TEST USING DOUBLE PRECISION ACCURACY.
C           IF ONE WISHES TO COMMUNICATE WITH KOLMO IN A DOUBLE
C           PRECISION PROGRAM, HE SHOULD CALL THE FORTRAN SUPPLIED
C           PROGRAM SNGL(X) PRIOR TO CALLING KOLMO, AND CALL THE
C           FORTRAN SUPPLIED PROGRAM DBLE(X) AFTER EXITING FROM KOLMO.
C           (NOTE THAT SUBROUTINE SMIRN DOES HAVE DOUBLE PRECISION
C           CAPABILITY AS SUPPLIED BY THIS PACKAGE.)
C
C
C        SUBROUTINES AND FUNCTION SUBPROGRAMS REQUIRED
C           SMIRN, NDTR, AND ANY USER SUPPLIED SUBROUTINES REQUIRED.
C
C        METHOD
C           FOR REFERENCE, SEE (1) W. FELLER--ON THE KOLMOGOROV-SMIRNOV
C           LIMIT THEOREMS FOR EMPIRICAL DISTRIBUTIONS--
C           ANNALS OF MATH. STAT., 19, 1948.  177-189,
C           (2) N. SMIRNOV--TABLE FOR ESTIMATING THE GOODNESS OF FIT
C           OF EMPIRICAL DISTRIBUTIONS--ANNALS OF MATH. STAT., 19,
C           1948.  279-281.
C           (3) R. VON MISES--MATHEMATICAL THEORY OF PROBABILITY AND
C           STATISTICS--ACADEMIC PRESS, NEW YORK, 1964.  490-493,
C           (4) B.V. GNEDENKO--THE THEORY OF PROBABILITY--CHELSEA
C           PUBLISHING COMPANY, NEW YORK, 1962.  384-401.
C
C     ..................................................................
C
      SUBROUTINE KOLMO(X,N,Z,PROB,IFCOD,U,S,IER)
      DIMENSION X(1)
C
C          NON DECREASING ORDERING OF X(I)'S  (DUBY METHOD)
C
      IER=0
      DO 5 I=2,N
      IF(X(I)-X(I-1))1,5,5
    1 TEMP=X(I)
      IM=I-1
      DO 3 J=1,IM
      L=I-J
      IF(TEMP-X(L))2,4,4
    2 X(L+1)=X(L)
    3 CONTINUE
      X(1)=TEMP
      GO TO 5
    4 X(L+1)=TEMP
    5 CONTINUE
C
C           COMPUTES MAXIMUM DEVIATION DN IN ABSOLUTE VALUE BETWEEN
C           EMPIRICAL AND THEORETICAL DISTRIBUTIONS
C
      NM1=N-1
      XN=N
      DN=0.0
      FS=0.0
      IL=1
    6 DO 7  I=IL,NM1
      J=I
      IF(X(J)-X(J+1))9,7,9
    7 CONTINUE
    8 J=N
    9 IL=J+1
      FI=FS
      FS=FLOAT(J)/XN
      IF(IFCOD-2)10,13,17
   10 IF(S)11,11,12
   11 IER=1
      GO TO 29
   12 Z =(X(J)-U)/S
      CALL NDTR(Z,Y,D)
      GO TO 27
   13 IF(S)11,11,14
   14 Z=(X(J)-U)/S+1.0
      IF(Z)15,15,16
   15 Y=0.0
      GO TO 27
   16 Y=1.-EXP(-Z)
      GO TO 27
   17 IF(IFCOD-4)18,20,26
   18 IF(S)19,11,19
   19 Y=ATAN((X(J)-U)/S)*0.3183099+0.5
      GO TO 27
   20 IF(S-U)11,11,21
   21 IF(X(J)-U)22,22,23
   22 Y=0.0
      GO TO 27
   23 IF(X(J)-S)25,25,24
   24 Y=1.0
      GO TO 27
   25 Y=(X(J)-U)/(S-U)
      GO TO 27
   26 IER=1
      GO TO 29
   27 EI=ABS(Y-FI)
      ES=ABS(Y-FS)
      DN=AMAX1(DN,EI,ES)
      IF(IL-N)6,8,28
C
C           COMPUTES Z=DN*SQRT(N)  AND  PROBABILITY
C
   28 Z=DN*SQRT(XN)
      CALL SMIRN(Z,PROB)
      PROB=1.0-PROB
   29 RETURN
      END
```

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 STUDY OBJECTIVES

The objectives of this study were to identify a set of characteristics of quality software and, for each characteristic, to define a metric such that:

1) Given an arbitrary program, the metric provides a quantitative measure of the degree to which the program has the associated characteristic, and

2) Overall software quality can be defined as some function of the values of the metrics.

Although "software" can have many components such as functional specifications, test plans, and data collection guidelines, this study concentrates on metrics which are applied to FORTRAN source programs (which may include extensive use of comment cards). However, many of the metrics in the report can be adapted straightforwardly to other software components.

1.2 STUDY APPROACH

1.2.1 <u>Initial Phase: Quick Quantification</u>

The study began by formulating a list of software characteristics, formulating a large number of quantities which could be quickly derived from a scan of a FORTRAN program and which appeared to have some correlation with software quality, and formulating an overall metric for software quality as a linear combination of the individual quantities.

1.2.2 <u>Evaluation of Quick Quantification</u>

Next, an evaluation of the results of the initial phase was performed. Several significant conclusions emerged.

1.2.2.1 For virtually all the simple quantitative formulas, it was easy to find counterexamples which eroded their credibility as indicators of software quality. Some examples are given below.

1) A metric was developed to measure program complexity in terms of the fraction of program statements which are branch statements. However, consider a program which reads some input, proceeds to

an m x m decision table, referring to (say) m separate tasks, each n statements long, followed by a printout and terminate. Excluding the read and print statements, the fraction of branch statements is:

$$FB = \frac{m^2}{m^2 + mn} = \frac{m}{m + n}$$

This program should be equally "good" for any reasonable values of m and n. However,

if $m = 10$ and $n = 1$, $FB = 0.91$

if $m = 3$ and $n = 30$, $FB = 0.09$

2) A metric was developed to calculate the average size of program modules as a measure of structuredness. However, suppose one has a software product with n 100-statement control routines and a library of m 5-statement computational routines, which would be considered well structured for any reasonable values of m and n, then, if $n = 2$ and $m = 98$, the average module size is 6.9 statements, while if $m = 10$ and $n = 10$, the average module size is 52.5 statements.

3) A metric was developed for the fraction of statements with potential singularities (divide, square root, logarithm, etc.) which were preceded by statements which tested and compensated for singularities. However, often the operation is in a context which makes the singularity impossible; a simple example is calculating the hypotenuse of a right triangle:

```
Z = SQRT(X**2 + Y**2)
```

4) Metrics were developed for the number of comment cards, the average length of comments, etc. However, it was fairly easy to recall programs with fewer and shorter comments which were much easier to understand than some with many extensive but poorly written comments.

1.2.2.2 The software field is still evolving too rapidly to establish metrics in some areas. In fact, doing so would tend to reinforce current practice, which may not be good. For example, some of the extra-program metrics included:

1) Existence of accompanying flow charts. However, with structured programming, these may not be needed.

2) Existence of test plan and conformity with (unit test, subsystem test, system test, integration) sequence. However, top-down programming performs these in parallel via stubs and a test skeleton.

3) Conformity of manpower plan with (46 percent analysis, 20 percent coding, 34 percent testing) breakdown. However, the use of top-down and other design validation procedures and the availability of automated test tools may reduce the testing effort a great deal.

1.2.2.3 In software product development and evaluation, one is generally far more interested in <u>where</u> and <u>how</u> rather than <u>how often</u> the product is deficient. Thus, the most valuable automated tools for software analysis would generally be those which flagged deficiencies or anomalies in the program rather than just producing numbers. This has, of course, been true in the past for such items as compiler diagnostics; one would be fairly irritated with a mere statement that "1.17 percent of your statements have unbalanced parentheses."

1.2.2.4 Calculating and understanding the value of a single, overall metric for software quality may be more trouble than it is worth. The major problem is that many of the individual characteristics of quality are in conflict: added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness can conflict with legibility. Users generally find it difficult to quantify their preferences in such conflict situations. Another problem is that the metrics are generally incomplete measures of their associated characteristic. To summarize these considerations:

1) The quality of a software product varies with the needs and priorities of the prospective user.

2) There is, therefore, no single metric which can give a universally useful rating of software quality.

3) At best, a prospective user could receive a useful rating by furnishing the rating system with a thorough set of checklists and priorities.

4) Even so, since the metrics are not exhaustive, the resulting overall rating would be more suggestive than conclusive or prescriptive.

5) Therefore, the best use for metrics at this point is as individual anomaly indicators, to be used as guides to software development, acquisition, and maintenance.

1.2.2.5 Most sets of software characteristics are too loosely defined and overlapping to be of much practical use*. This led to a further effort to define more precisely the set of characteristics and their interrelations with each other, and with a refined set of quality metrics.

1.2.3 Final Phase: Hierarchical Characteristics and Anomaly-Detecting Metrics

Based on the conclusions above, the study proceeded along the following lines to develop a hierarchical set of characteristics and a set of anomaly-detecting metrics.

1) Define a set of characteristics which are important for software, and reasonably exhaustive and nonoverlapping.

2) Develop candidate metrics for assessing the degree to which the software has the defined characteristic.

3) Develop a set of "characteristics of quality metrics," including such items as correlation with software quality, magnitude of potential benefits of using, quantifiability, ease of automation.

4) Evaluate each candidate metric with respect to the above criteria, and with respect to its interactions with other metrics: overlap, dependencies, shortcomings, etc.

5) Based on these evaluations, refine the set of software characteristics into a set which is more mutually exclusive and exhaustive with respect to the uses of software quality evaluation.

6) Define the metrics and reorganize them with respect to the primitive characteristics which resulted from Step 5.

7) Develop detailed algorithmic forms for the metrics.

Many of the resulting anomaly-detecting metrics involve some degree of source program text analysis and structure analysis. Although many of these analyses are straighforward, they are sufficiently complex to make their detailed development a job too large for the scope of this small study. Thus, in Step 7, the study proceeded to detailed algorithms

*One exception is the set given by Wulf in his "Report of Workshop #3" for the Monterey Symposium on the High Cost of Software, September 1973.

only in one area, that of header commentary; elsewhere, the type of automated evaluation (algorithm or compliance checker) is indicated, along with an assessment of the ease of developing the evaluation program and the degree of completeness of the resulting evaluation.

1.3 MAJOR RESULTS

The major results of the study are the following:

1) An assessment of the limitations of purely quantitative measures of software quality, given in Section 1.2.2 above.

2) A definitive hierarchy of characteristics of software quality, the "Characteristics Tree," developed in Section 3.0.

3) An extensive list of anomaly-detecting metrics, thoroughly classified with respect to the characteristics of software quality and evaluated with respect to a set of "Characteristics of quality metrics." These metrics are developed in Sections 4.1 through 4.4 from an initial list given in the Appendix.

4) A detailed algorithm and guidelines for using metrics to perform assessment of program header comments is given in Section 4.5.

5) A discussion and cost-benefit analysis of the potential impact of using anomaly-detecting metrics during the software process. This is contained in Section 5.0, based on the descriptive outline of the software development process given in Section 2.0.

6) An annotated bibliography of relevant literature, given in Section 6.0.

The study's most important contribution though, has been to provide, for the first time, a clear, well-defined framework for assessing the often slippery issues associated with software quality, via the consistent and mutually supportive sets of definitions, distinctions, guidelines, and experience summaries cited above. This framework is certainly not complete, but it has been brought to a point sufficient to support the evaluation of the relative cost-effectiveness of prospective code-analysis tools presented in this report, and to serve as a viable basis for future refinements and extensions.

1.4 USEFUL DIRECTIONS FOR FURTHER RESEARCH

Based on the foundation established in this report, a number of research projects could be undertaken which would be likely to produce useful and significant results. These include:

1) Detailed design and development of computer programs to perform the evaluation indicated by the anomaly-detecting metrics.

2) Application of the resulting programs to software development and software package procurement efforts, followed by an evaluation and refinement of the metrics.

3) Adaptation of the metrics into checklists to aid software analysts in assessing the design implications of various software requirements.

4) Extension of the metrics to cover other software products besides the source code.

5) Establishing a continuing service for the accumulation and dissemination of information on software metrics and experience in using them.

2.0 SOFTWARE AND THE SOFTWARE DEVELOPMENT PROCESS

In this report, software is defined as computer program code and its associated necessary data and documentation. The Software Development Process is the formal process by which program objectives are transformed into program requirements, then into design specifications, implemented into code, tested, and finally placed and maintained in operational status. The degree of formality of the Software Development Process depends upon the purpose and end use of the software and also upon the size of the software development team involved. At one extreme lies the one-man, one-shot design analysis computer program developed in the informal environment of a timeshare terminal. At the other extreme lies the critical real-time program requiring thousands of man-hours to develop. Obviously, the degree of formality required in the Software Development Process differs radically between these two extremes. This section of the report treats the formal Software Development Process.

The relationship between quality software and the Software Development Process is analogous to that between quality hardware and what might be termed the "Hardware Development Process." Just as experience dictated the need for formal development processes in order to insure quality hardware, experience has shown the need for formal processes in the development of quality software. (The tendency to continually modify software creates an added dimension of difficulty in the software case, however.) In essence, it is not enough just to only examine the end item, be it hardware or software, in order to judge its quality. It is not feasible, within the usual constraints of time and resources, to test the end item in all modes of operation, in all environments, or for all conditions under which it is expected to perform. Instead, judicious review and testing of the software products generated within the framework of the formal Software Development Process is the best application of available resources.

2.1 THE SOFTWARE DEVELOPMENT PROCESS

The Software Development Process may be divided into a number of more or less independent development steps or phases. Just how many steps and just what is carried out within those steps is to some degree dependent upon the purpose and size of the software being developed. The software development steps involved in the delivery of a large software system to a customer is illustrated in Figure 2-1. Each of these development steps produces software products or makes specific contributions to software products. The principal software products are shown in Figure 2-2, where it will be noted that software products are of two types: documentation and computer program code. Certain groupings of these software products form the basis for formal technical reviews. Upon approval, some of the software products are "baselined" and serve as the basis for configuration management. The relationships among the software products, the technical reviews, and the baselines are summarized in Table 2-1.

The primary function of software configuration management is the control of changes to established baselines, not only to customer-recognized baselines, such as the Allocated and Product Baselines, but to project internal baselines as well, such as Design Review and Formal Test Baselines. Software configuration management performs other functions as well, including control of configuration identification, maintaining configuration status records, and monitoring configuration reviews and audits.

In summary, the formal Software Development Process is organized into a number of steps, each of which produces or contributes to the production of certain specific software products. Formal review and audits of certain software products lead to the establishment of "baselines," and these baselines serve as the basis for configuration management, which is so crucial in the development of quality software.

The next section further defines the principal software products, while subsequent sections will deal with software quality aspects.

2.2 THE SOFTWARE PRODUCTS

For the discussion of this section, it is assumed that the software being developed is a subsystem of a larger entity and that a systems requirements specification has been previously formulated and baselined. It is further assumed that this systems requirements specification has allocated to the Software Subsystem the top-level functional, performance, and design requirements that the Software Subsystem is expected to meet

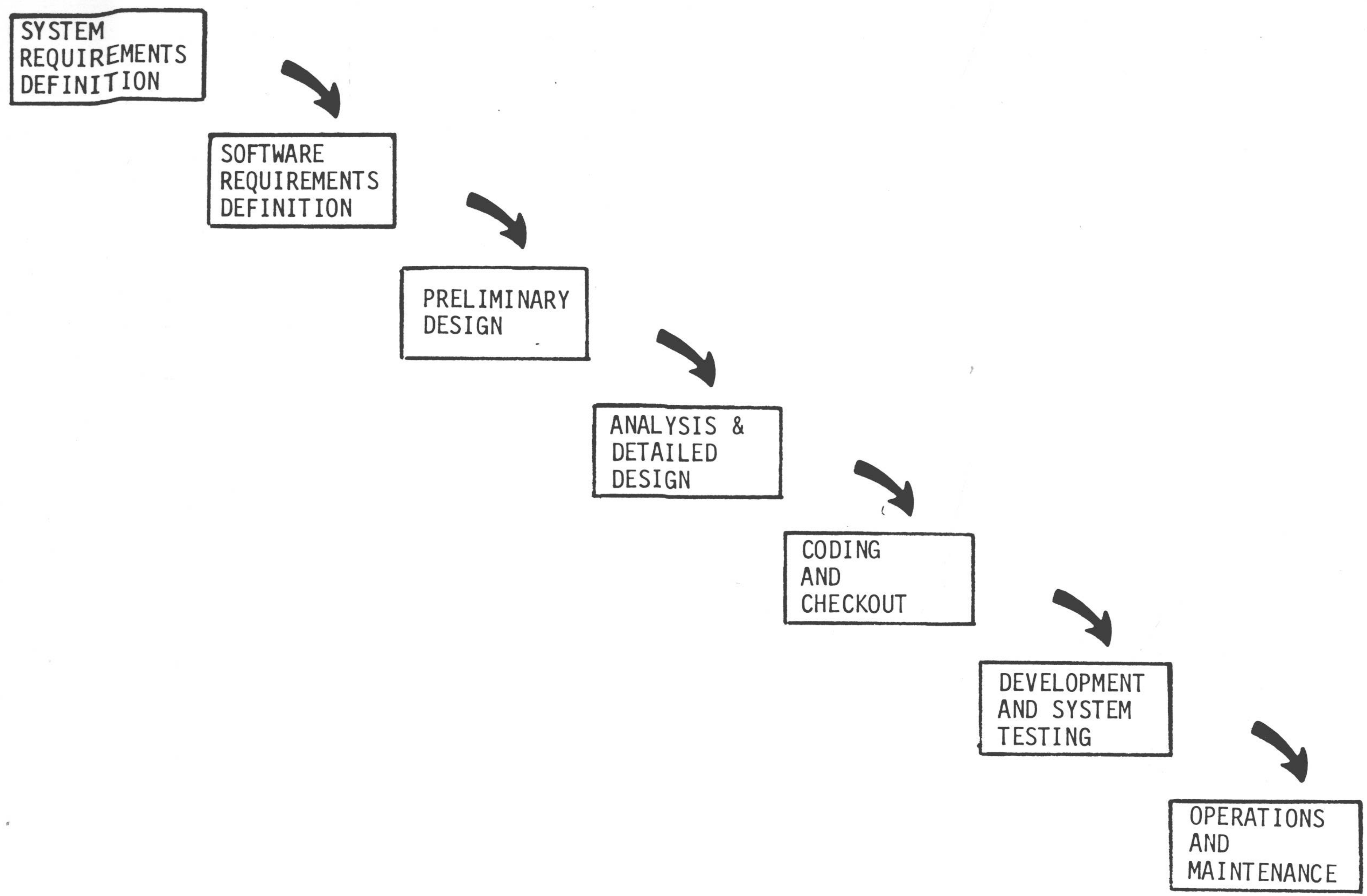


Figure 2-1. Software Development Steps for Large-Scale Software