Constructing and Testing Software Maintainability Assessment Models

Fang Zhuo

IBM Advanced Workstation Division
Austin, Texas

Bruce Lowther, Paul Oman & Jack Hagemeister

Software Engineering Test Lab
University of Idaho
Moscow, Idaho 83843
Contact: oman@cs.uidaho.edu

Abstract

Software metrics are used to quantitatively characterize the essential features of software. This paper investigates the use of metrics in assessing software maintainability by presenting and comparing seven software maintainability assessment models. Eight software systems were used for initial construction and calibrating the automated assessment models, and an additional six software systems were used for testing the results. A comparison was made between expert software engineers' subjective assessment of the 14 individual software systems and the maintainability indices calculated by the seven models based on complexity metrics automatically derived from those systems. Initial tests show very high correlations between the automated assessment techniques and the subjective expert evaluations.

1 Introduction

Implementing sound maintenance changes and constructing maintainable software is complicated by the many aspects of software that affect maintenance activity. Characteristics such as the operational environment, supporting documentation and the attributes of the source code itself, all influence maintenance effort. We define *maintenance* as the process of implementing corrective, adaptive or perfective software changes (a definition adapted from Lientz and Swanson [11] and consistent with [2, 5, 7, 8, 9, 10, 12, 26, 27, 28, 31, 32]). Further, we define *maintainability* as the characteristics of the software and associated environment that affect the maintenance process and are indicative of the amount of effort necessary to perform maintenance changes [1, 4, 2, 6, 13, 14, 23, 25, 28, 29]. Determining the maintainability of a software system is, therefore, a multidimensional assessment. Although there are methods and instruments for subjectively assessing software maintainability, few studies have looked into automating the process and fewer still have compared subjective maintainability assessment with the results obtained from automated means.

In a study of 35 published works on software maintainability, we coalesced software maintainability definitions into a hierarchical structure containing 92 known maintainability attributes [19]. This *software maintainability taxonomy* has facilitated the construction of mathematical models and tools for assessing software maintainability. Software metrics are used to quantitatively characterize the attributes within the taxonomy. Essential features of the software can then be measured via a battery of metrics and combined into a single value (hybrid metric) representing the maintainability of the software being evaluated. Of all of the factors influencing maintenance, however, the most commonly cited and modified component of the software is the source program itself. Hence, this paper focuses primarily on assessing software maintainability based on features of the source program.

The ultimate use of software maintainability measures may be applied in industrial environments in three ways: (1) As a managerial assessment to quantify the cost of maintaining existing software systems, (2) As a quality assessment and control mechanism to drive software development efforts, and (3) As a mechanism to enforce maintainability standards prior to acceptance and/or delivery.

2 Software Maintainability Models

One of the earliest maintainability assessment techniques was defined by Berns [2] who created a Maintainability Analysis Tool (MAT). Berns' tool calculates the maintainability of a FORTRAN program by summing the difficulty indices of the program elements adjusted by a system of weights. By tuning the vector of weights, he adjusted the calculated maintainability index. Although Berns' tool was never validated with industrial code his notion of calculating an "index of difficulty" is both interesting and useful. Several of the methods described in this paper use this technique.

Using a subset of the metrics defined in our software maintainability taxonomy, we conducted a series of experiments to determine which metrics were leading indicators of maintainability and how they could be combined into a single index of maintainability. Towards this goal we implemented and tested an automated maintainability assessment tool, con-

0-8186-3740-4/93 $3.00

structed three non-linear polynomial regression models of maintainability, developed a model of maintainability based on entropy, and constructed models based on factor analysis and principal component analysis. Each of the following subsections introduces a type of maintainability assessment model. A comparison of all seven models is given in Section 3.0.

2.1 Hierarchical Multidimensional Assessment Model

A multidimensional assessment evaluates the maintainability of a complicated software product by categorizing its characteristics into hierarchical levels. Each branch beneath the topmost level (the root) is a dimension. Each dimension is subsequently divided and characterized in a lower level of the hierarchy. Successive refinements of the hierarchy are constructed until an atomic metric is identified and defined. Then a method called *Weight and Trigger Point Range analysis* (adapted from [24]) is used to quantify maintainability by calculating "degree of fit" from a table of acceptable metric ranges.

For this particular type of maintainability problem Oman [18, 19] has suggested a hierarchical model that divides the characteristics into three dimensions:

1. *Control structure*: Includes characteristics pertaining to the manner in which the program or system is decomposed into algorithms.
2. *Information structure*: Includes characteristics pertaining to the choice and use of data structure and data flow techniques.
3. *Typography, Naming, and Commenting*: Includes characteristics pertaining to the typographic layout, naming and commenting of code.

These three dimensions make up the highest level in the hierarchy of source code maintainability. Their contributions to overall maintainability are measured by more specific atomic metrics at a lower level. Once the atomic metric attributes are defined for each dimension, they can be combined to calculate maintainability one level up the hierarchy. This principle had been implemented in a prototype maintainability assessment system (HP-MAS) developed for Hewlett-Packard by the University of Idaho Software Engineering Test Lab [22].

Some metric attributes are more critical to maintainability than others, to reflect this importance a weight is introduced and attached to each metric attribute. This provides a method of emphasizing more important metric attributes by using a more significant weight. In all of our experiments all HP-MAS weights were set to unity.

Most metrics have an optimum range of values within which the software is more easily maintained. When the metric value falls outside the optimum range, it indicates the maintainability is lower, or there is a deviation (or penalty) on its contribution to maintainability. The optimum range value, called the *trigger point range*, reflects the "goodness" of the program style. For example, if the trigger point for *average lines of code* (aveLOC) in each module is 5 to 75, software in this range may be thought as having good style. Otherwise it may be classified as poor style. If the metric value lies between the upper and lower bound (trigger point range), there is no deviation. Otherwise, if the metric value falls outside the trigger point range, but is close to the bounds, then there exists a proportional deviation which can run up to 100 percent (the maximum penalty). The weighted deviation is computed by multiplying the weight factor times the calculated deviation. The metric attributes are combined based on the tenet that the dimensional maintainability is assumed to be 100% (highly maintainable), then reduced by the percent deviation of each metric. The dimension maintainability, $DM_{dimension}$, is calculated as:

$$DM_{dimension} = 1 - \frac{\sum(w_i D_i)}{\sum w_i}$$

And the overall maintainability index is a combination of the three dimensions:

$$\begin{aligned} Maintainability &= 100 * DM_{control} \\ &\quad * DM_{information} \\ &\quad * DM_{typography} \end{aligned}$$

Multiplying the three dimension's maintainability gives a lower overall maintainability than averaging would, illustrating that deviation in one aspect of maintainability will hinder other aspects of the maintenance effort, thus reducing the maintainability of the entire system. Our tests show that this method of maintainability assessments can be tuned to give highly accurate maintainability indices.

2.2 Polynomial Regression Models

Regression analysis is a statistical method for predicting values of one or more response (dependent) variables from a collection of predictor (independent) variables. In order to assess software maintainability, one approach is to create a polynomial equation where the maintainability of a system is expressed as a function of the associated metric attributes. We have used this technique to develop a set of nonlinear polynomial maintainability assessment models [21].

The classical linear regression model for n independent observations on Y with the associated predictor variables of $x_i (i = 1, \ldots, p)$, is the set of polynomials:

$$\begin{aligned} Y_1 &= \beta_0 + \beta_1 x_{11} + \cdots + \beta_p x_{1p} + \varepsilon_1 \\ Y_2 &= \beta_0 + \beta_1 x_{21} + \cdots + \beta_p x_{2p} + \varepsilon_2 \\ &\vdots \qquad\qquad\qquad\qquad \vdots \\ Y_n &= \beta_0 + \beta_1 x_{n1} + \cdots + \beta_p x_{np} + \varepsilon_n \end{aligned}$$

The model uses a set of values of Y and $x_1, x_2, \ldots, x_p$ to estimate regression coefficients $\beta_0, \ldots, \beta_p$ and the error $\varepsilon_1, \ldots, \varepsilon_n$. Values for $\beta_0, \ldots, \beta_p$ should be chosen in such a way that the sum

of the squares of differences between the observed values Y and estimated values $\hat{Y}$ by the regression equation is as small as possible. Ideally, the multivariate regression model requires that the predictor variables be independent. However, the software complexity metrics contained a high degree of collinearity. Reducing the number of metrics will improve the performance of the regression model when high collinearity exists in the metrics. Using several metrics (Halstead's metrics, McCabe's cyclomatic complexity, lines of code, lines of comments, number of executable semicolons, average variable span, number of blank lines, number of tokens, lines of data declarations, level of control structure nesting, and others) as indicators of the complexity and/or quality inherent in each of the code samples, a series of correlations and principal component analyses were conducted. We wanted to determine which metrics were computationally redundant with other metrics in the set, and which metrics were leading indicators of maintainability (as measured by the initial survey). In spite of the current research trend away from the use of Halstead metrics, all of our tests clearly indicated that Halstead's metrics (specifically Volume and Effort) were the best predictors of maintainability for our test data. Our correlations and principal component analyses suggested that:

1. The Halstead metrics for program Effort and Volume were very strong predictors of maintainability.
2. Only a single Halstead metric is necessary – either Effort, Volume, Length or Predicted Length.
3. The Halstead primitive metrics (η_1, η_2, N_1, and N_2) added little that was not accounted for in Volume and Effort.
4. The two cyclomatic complexity metrics were equally significant. Either the simple cyclomatic complexity or the extended cyclomatic complexity can be used for modeling maintainability with equal effectiveness
5. The number of lines of code, purity ratio, variable span, and number of comments contribute to the overall picture of maintainability.
6. The number of blanks, tokens, data declarations, arguments, and maximum level of nesting contributed little to the prediction of maintainability for this set of data.
7. Module averages, rather than total metric values, tend to be more stable and, therefore, more indicative of the true nature of the maintainability of the software suite.

Approximately 50 regression models were constructed and tested in our attempts to identify simple models which could be calculated from existing tools and still be generic enough to be applied to a wide range of software. We constructed regression models with R^2 values ranging from a low of 0.60 to a high of 0.998. The R^2 value indicates the proportion of variation within the data set which is explained by the model. A R^2 value of 1.0 is perfect, while R^2 values greater than 0.80 are considered very good. Although we derived models with R^2 values exceeding 0.99, it was clear from the makeup of those models that they were specific to the test data set and would not be applicable to a wider range of software. The three best regression models were:

1. A single metric model based on Halstead's Effort.

$$Maintainability = 125 - 10 * \log(aveE)$$

Where aveE is the average Halstead Effort per module.

2. A four metric polynomial based on Halstead's Effort, Extended cyclomatic complexity, Lines of code, and Number of comments.

$$\begin{aligned} Maintainability = {} & 171 - 3.42 * \ln(aveE) \\ & -0.23 * aveV(g') \\ & -16.2 * \ln(aveLOC) \\ & +0.99 * aveCM \end{aligned}$$

Where, aveE is the same as in the single metric model, aveV(g′) is the average extended cyclomatic complexity per module, aveLOC is the average lines of code per module, aveCM is the average number of lines of comment per module.

3. A five metric linear regression model based on Halstead's Effort, Extended cyclomatic complexity, Lines of code, Number of comments, and a subjective evaluation of the external documentation and complexity of "building" the software.

$$\begin{aligned} Maintainability = {} & 138 - 2.76 * \ln(aveE) \\ & -0.33 * aveV(g') \\ & -12.2 * \ln(aveLOC) \\ & +0.88 * aveCM \\ & +1.04 * EDOC \end{aligned}$$

Where, aveE, aveV(g′), aveLOC and aveCM are the same as in the four metric model and EDOC is a 15 point subjective evaluation of the external documentation plus a 5 point subjective evaluation of the ease of building the system.

Since maintenance is influenced by more factors than just the source code of the software system, a single subjective metric (EDOC) which reflects the documentation and the ease of building the software was added in an attempt to more accurately compute the maintainability. Our tests show that all three models compute reasonably accurate maintainability scores.

2.3 Estimating Maintainability using Complexity

Another approach, used by Munson and Khoshgoftaar [15, 16, 17], has been suggested to address maintainability via complexity, a quantitative measurement of the difficulty that a programmer encounters when working with source code. Two software complexity evaluation methods developed by Munson and Khoshgoftaar, complexity in terms of entropy and complexity in terms of underlying factors, are presented here.

2.3.1 Software Complexity Analysis via Entropy

The concept of complexity measured by entropy was introduced first by Van Emden [30] who defined the complexity as "the way in which a whole is different from the composition of its parts." Based on this definition, Munson and Khoshgoftaar [17] suggested that if a software system can be divided into a set of k functional modules, and if each module's complexity can be measured by n metrics $M_1, \ldots, M_n$, there will be n complexity measures for the k modules of the system represented by a $(k \times n)$ matrix, and the informational measure of dependence between $M_1, \ldots, M_n$, called *interaction*, is directly related to the complexity of the system. According to information theory, the interaction of $M_1, \ldots, M_n$ can be represented by:

$$R(M_1, \ldots, M_n) = H(M_1) + \cdots + H(M_n) - H(M_1, \ldots, M_n)$$

Where, $H(M_i)$ is the entropy of the i^{th} metric and $H(M_1, \ldots, M_n)$ is the joint entropy of metrics.

Assuming metrics $M_1, \ldots, M_n$ have normal probability distributions, then each metric entropy and the joint entropy can be represented as:

$$H(M_i) = \frac{1}{2}\ln(2\pi) + \frac{1}{2}\ln(\sigma_{ii}) + \frac{1}{2}$$

$$H(M_1, \ldots, M_n) = \frac{n}{2}\ln(2\pi) + \frac{1}{2}\ln(|\sum|) + \frac{n}{2}$$

Where, $\sum$ is the covariance matrix of $M_1, \ldots, M_n$, $|\sum|$ is its determinant, σ_{ii} is the variance of M_i or the i^{th} diagonal element of $\sum$, which is a real symmetric matrix. Hence, $|\sum|$ can be expressed as:

$$|\sum| = \prod_{i=1}^{n} \lambda_i$$

Where the λ_i is an eigenvalue of the covariance matrix, $\sum$, yielding:

$$R(M_1, \ldots, M_n) = \frac{1}{2}\sum_{i=1}^{n}\ln(\sigma_{ii}) - \frac{1}{2}\ln(\prod_{i=1}^{n}\lambda_i)$$

The software complexity is defined as the maximum of the interaction, $R(M_1, \ldots, M_n)$, which is attained when

$$\sigma_{11} = \sigma_{22} = \cdots = \sigma_{nn} = \frac{1}{n}trace(\sum).$$

Therefore the system complexity, C_i, can be represented as:

$$C_i = -\frac{1}{2}\sum_{i=1}^{n}\ln(\frac{\lambda_i}{\overline{\lambda}})$$

Where, λ_i is an eigenvalue of the covariance matrix $\sum$ and $\overline{\lambda}$ is the average eigenvalue.

To avoid the infinite value caused by zero eigenvalues, the software complexity is approximated in terms of only positive eigenvalues $\lambda_1, \ldots, \lambda_m$ of $\sum$, where $m \leq n$. That is:

$$C_i \approx -\frac{1}{2}\sum_{i=1}^{m}\ln(\frac{\lambda_i}{\overline{\lambda}})$$

Where, $\overline{\lambda}$ is the average of the positive eigenvalues and C_i is the software system interaction complexity as defined by Munson and Khoshgoftaar [17].

To convert the results of the complexity calculation into a quantitative maintainability value, a simple expression was used to normalize the value around 80 to bring it into alignment with our other assessment models. Hence,

$$Maintainability = 80 - C_i$$

which reflects the inverse relationship between maintainability and complexity.

For our tests, fifteen metrics were calculated for every individual module in each of several software systems. The k modules of each system were represented by a $(k \times 15)$ matrix. The measured metrics included Halstead's unique operator count (η_1), unique operand count (η_2), total operator count (N_1), total operand count (N_2), Length (N), predicted Length ($\hat{N}$), Purity ratio (P/R), Volume (V), Effort (E), McCabe's complexity ($V(g)$), extended complexity ($V(g')$), lines of code (LOC), lines of comments (CM), number of executable statements (ES), and average number of statements between two successive references to the same variable (SP). The eigenvalues of the metrics' covariance matrix and the interaction complexity, C_i, for each system were then calculated. After several iterations of testing, we found that the maintainability model was too sensitive to the metric values. That is, the maintainability result differed widely with slight deviations in the metrics. The 15 metrics were then grouped into subsets and a series of tests were conducted against each subset. The result is that a five metric subset consisting of V, V(g'), LOC, CM and SP proved to give us the best normalized maintainability index. Our tests show that this simplified model calculates significantly accurate maintainability indices.

2.3.2 Software Complexity via Underlying Factors

In the previous subsection, the system's complexity is aggregated by analyzing the interaction between metrics. Munson and Khoshgoftaar's other method uses a statistical factor analysis to convert the metrics into a few underlying, but unobservable quantities called *factors*. The complexity model is then constructed in terms of these few underlying factors.

Given a set of n metrics, $M = (M_1, \ldots, M_n)$, with mean, $\mu = (\mu_1, \ldots, \mu_n)$, and covariance matrix $\sum$, factor analysis seeks to group the metrics by explaining the covariance structure in such a way that the metrics within a particular group are highly correlated, but with relatively small correlations to metrics in different groups. Each group of metrics represents a single underlying factor. The model of the factor analysis can be represented as:

$$\begin{aligned} M_1 - \mu_1 &= l_{11}F_1 + l_{12}F_2 + \cdots + l_{1m}F_m + \varepsilon_1 \\ M_2 - \mu_2 &= l_{21}F_1 + l_{22}F_2 + \cdots + l_{2m}F_m + \varepsilon_2 \\ &\vdots \qquad\qquad\qquad\qquad \vdots \\ M_n - \mu_2 &= l_{n1}F_1 + l_{n2}F_2 + \cdots + l_{nm}F_m + \varepsilon_n \end{aligned}$$

And the vectors $\mathbf{F}$ and $\boldsymbol{\varepsilon}$ must satisfy:

$$\begin{aligned} E(\mathbf{F}) = \mathbf{0}, \quad Cov(\mathbf{F}) = \mathbf{I} \\ E(\boldsymbol{\varepsilon}) = \mathbf{0}, \quad Cov(\boldsymbol{\varepsilon}) = \Psi \end{aligned}$$

Where, Ψ is a diagonal matrix, $F_1, \ldots, F_m$ are factors, and $\varepsilon_1, \ldots, \varepsilon_n$ are errors.

The coefficient l_{ij}, shows the loading of the i^{th} metric on the j^{th} factor. In other words, a high magnitude of l_{ij} means the j^{th} factor is highly related to the i^{th} metric. The metrics which have high loading coefficients to a common factor are grouped together, representing the factor. For example, if *Lines of code* (LOC) and *number of executable statements* (NES) have high loading coefficients to a same factor, then LOC and NES are grouped together by the factor analysis model and the factor may be interpreted as a "size" factor.

The factors are unobservable, but their values, called *factor score*, can be estimated by:

$$\mathbf{F} = \mathbf{Dz}$$

Where, $\mathbf{z}$ is the standardized metrics vector. That is:

$$z_i = \frac{(M_i - \mu_i)}{\sigma_i} \qquad (i = 1, \ldots, n)$$

Where μ_i and σ_i are the mean and standard deviation of metric M_i, respectively.

$\mathbf{D}$ is called the factor score coefficient matrix. The factor score coefficient matrix is constructed in such a way that it maps the standardized metrics vector $\mathbf{z}$ onto the underlying orthogonal factor dimensions. The goal is to determine the number of factors. The proportion of the covariance structure that can be explained increases with the number of factors in the model. It is preferable to use just a few factors, but enough are needed to explain the covariance structure.

A set of software systems is represented by a set of n metrics $M = (M_1, \ldots, M_n)$ with mean $\mu = (\mu_1, \ldots, \mu_n)$ and covariance matrix $\sum$. The Munson and Khoshgoftaar [15] factor complexity model for the set of p software systems is defined as:

$$\begin{aligned} C_{r1} &= f_{11}\lambda_1 + \ldots + f_{1m}\lambda_m \\ C_{r2} &= f_{21}\lambda_1 + \ldots + f_{2m}\lambda_m \\ &\vdots \qquad\qquad \vdots \\ C_{rp} &= f_{p1}\lambda_1 + \ldots + f_{pm}\lambda_m \end{aligned}$$

Where, C_{ri} represents the relative complexity of i^{th} system in the systems set, f_{ik} is the factor score for the i^{th} program on the k^{th} factor and λ_k is the associated eigenvalue. The variance of the complexity model, $V(\mathbf{C_r})$, can be represented as:

$$V(\mathbf{C_r}) = \boldsymbol{\lambda} Cov(\mathbf{F})\boldsymbol{\lambda}' = \sum_{i=1}^{m} \lambda_i^2$$

Where, $\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_m]$, λ_i is the eigenvalue associated with the i^{th} factor and m is the number of factors. In this context, the relative complexity model explains variance in proportion to the amount of total variance contributed by metrics $\mathbf{M}$.

From the premise that maintainability is inversely proportional to complexity, the maintainability model via complexity may be defined as:

$$Maintainability = 80 - \frac{10C_{ri}}{\sqrt{V(\mathbf{C_r})}}$$

Where, C_{ri} is the complexity of the software system and $V(\mathbf{C_r})$ is the variance of the factor complexity model as defined by Munson and Khoshgoftaar [15].

The maintainability score is again normalized to a value around 80 to bring it into alignment with our other assessment models.

There is a requirement in factor analysis that the number of variables (metrics) should be less than the number of observations (systems), in order to estimate the factor scores. Because of the limited number of available test systems, six metrics were chosen to measure each system. The six metrics include average Effort (aveE), average extended cyclomatic complexity (aveV(g')), average lines of code (aveLOC), average lines of comment (aveCM), average number of executable statements (aveES), and average variable span (aveSP). These six metrics were thought to be significant metrics gauging the software system maintainability.

Note that the factor analysis model can be rewritten as:

$$\mathbf{M} = \boldsymbol{\mu} + \mathbf{LF} + \boldsymbol{\varepsilon} = \boldsymbol{\mu} + \mathbf{LTT'F} + \boldsymbol{\varepsilon} = \boldsymbol{\mu} + \mathbf{L^*F^*} + \boldsymbol{\varepsilon}$$

Where, $\mathbf{T}$ is an orthogonal matrix so that $\mathbf{TT'} = \mathbf{I}$ $\mathbf{L^*} = \mathbf{LT}$ and $\mathbf{F^*} = \mathbf{T'F}$.

This means that both *loading* and *factor score* are not unique. This provides the rationale for *factor rotation*, which may give a better interpretation of factor structure. In other words, the objective of rotation is to make the pattern of loadings such that each metric loads highly on a single factor and has small-to-moderate loading on the remaining factors. Table 1 shows the rotated pattern of loading from which two distinct factors merged. The first factor is related to metric average Effort (aveE), average extended cyclomatic complexity (aveV(g')), and average executable statements (aveES), which may be interpreted as the "control" factor. While, the metrics average lines of code (aveLOC), average lines of comment (aveCM) and average variable span (aveSP), appear more related to the second factor, which might be called "size/data flow" factor.

Table 1: Rotated Factor Loading

Metric	Factor1	Factor2
E	**0.98089**	0.14064
V(g')	**0.93831**	0.31574
LOC	0.44933	**0.86550**
CM	0.04298	**0.99327**
ES	**0.93542**	0.33363
SP	0.48976	**0.84813**
Eigenvalues	4.5344	1.3126

Table 2 shows the factor score coefficient matrix, which is used to convert the standardized metrics into the underlying two factors which were used to calculated the maintainability index defined above. Our test calculations with this model were disappointing (non-significant), but that may be due to our limited test sample size. (See Section 3.0 for a description of our test results.)

Table 2: Standardized Scoring Coefficients

Metric	Factor1	Factor2
E	0.40498	-0.18788
V(g')	0.33890	-0.08348
LOC	-0.02893	0.33941
CM	-0.24650	0.51605
ES	0.33281	-0.07321
SP	-0.00603	0.31937

2.4 Principal Components Model

Principal component analysis is another statistical technique used to reduce collinearity between software metrics and reduce the number of *components* used to construct maintainability models. Principal component analysis orthogonalizes the metrics into the new components, called *principal components*. The components containing the most information are then selected to construct a regression maintainability model. Maintainability is expressed as a function of the principal components, instead of the original metrics. The technique is based on the covariance between original variables (metrics) $M_1, M_2, \ldots, M_n$, to linearly combine them in order to generate new components $Y_1, Y_2, \ldots, Y_n$. Geometrically, the new set of n components represent a new coordinate system obtained by rotating the original system with $M_1, M_2, \ldots, M_n$ as the coordinate axes. The new axes represent the directions with maximum variability and provide a simpler description of the covariance structure. That is, the first axis displays the largest variance, the second (orthogonal) axis, the next largest variance, and so on.

Let ρ be the correlation matrix associated with the original variable vector $M = [M_1, M_2, \ldots, M_n]$ and has the eigenvalue-eigenvector pairs $(\lambda_1, \mathbf{e_1}), (\lambda_2, \mathbf{e_2}), \ldots, (\lambda_n, \mathbf{e_n})$ where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$. The i^{th} principal component is defined as:

$$Y_i = \mathbf{e_i}'Z = e_{1i}Z_1 + e_{2i}Z_2 + \cdots + e_{ni}Z_n, \quad i = 1, 2, \ldots, n$$

Where, $Z_i (i = 1, \ldots, n)$ are standardized variables such that:

$$Z_i = \frac{(M_i - \mu_i)}{\sigma_i}$$

and μ_i and σ_i are mean and standard deviation of variable, M_i, respectively. With these choices:

$$Var(Y_i) = \lambda_i \qquad i = 1, 2, \ldots, n$$

$$Cov(Y_i, Y_k) = 0 \qquad i \neq k$$

And the proportion of total variance due to the k^{th} principal component, $P_k (k = 1, \ldots, n)$, is

$$P_k = \frac{\lambda_k}{\lambda_1 + \lambda_2 + \cdots + \lambda_n}$$

Where, the λ_i's are the eigenvalues of correlation matrix, ρ, and $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$.

If most of the total variance (for instance, 80% to 90%) can be attributed to the first few principal components, then these principal components can replace the original n variables without much loss of information. Because the principal components are orthogonal this reduces the collinearity in constructing regression models.

Principal components can be interpreted as to what they may represent. For example, if one principal component has a high magnitude coefficient with metrics *Lines of code, Number of operators* and *Number of operands*, and low magnitude coefficient with other metrics, then this principal component may be interpreted as the "size" component. The burden of interpretation lies on the observer's knowledge and is not intrinsic to the principal component analysis.

For our tests each software system was represented by the metrics aveη_1, aveη_2, aveN_1, aveN_2, aveN, ave$\hat{N}$, aveP/R, aveV, aveE, ave$V(g)$, ave$V(g')$, aveLOC, aveCM, aveES and aveSP. Using principal

component analysis, these metrics are orthogonalized into their principal components. Table 3 shows the proportion of variance information each principal component contained. This table indicates that the first three principal components contain 98% of the variance. Therefore, the maintainability model was constructed using only the first three principal components. The principal components maintainability model was calculated as:

$$Maintainability = 78.62 - 1.90 * PRIN_1 + 5.75 * PRIN_2 + 5.62 * PRIN_3$$

Where, $PRIN_1, PRIN_2$ and $PRIN_3$ are the first, second, and third principal components respectively.

Analysis of the eigenvectors associated with the first three principal components indicates that the first principal component is roughly a "control/volume" component, the second principal component is mainly related to the metric *Average lines of comments*, and the third principal component is mainly dependent on the metric *Average Purity ratio*. Our tests show that this model calculates significantly accurate maintainability indices.

Table 3: Eigenvalues and Proportion Variance

Principal	Eigenvalue	Proportion	Cumulative
$PRIN_1$	11.7850	0.785665	0.78567
$PRIN_2$	1.89416	0.160730	0.94639
$PRIN_3$	0.5168	0.034453	0.98085
$PRIN_4$	0.1730	0.011531	0.99238
$PRIN_5$	0.0678	0.004522	0.99690
$PRIN_6$	0.0434	0.002895	0.99980
$PRIN_7$	0.0031	0.000204	1.00000

3 Testing the Models

Four approaches to maintainability assessment were applied to construct seven models. All four approaches (and models) were applied to the same data and their performance was compared. The approaches and assessment techniques are summarized as follows:

- **Hierarchical Multidimensional Assessment**
  The software maintainability is modeled as a hierarchical structure of complex attributes pertaining to the software system. The maintainability is quantified by traversing the hierarchy and calculating the weighted deviation of each dimension.

- **Polynomial Regression Models**
  This method uses statistical regression analysis as a tool to explore the relationship between software maintainability and software metrics. Three polynomial regression models were constructed to assess software maintainability.

- **Estimating Maintainability via Complexity**
  Software complexity factor analysis and entropy was used to gauge software maintainability via software complexity. Two maintainability models were constructed using *aggregate* and *relative* complexity evaluation methods.

- **Principal Components Maintainability Model**
  Principal component analysis was used to eliminate metric collinearity in order to create a stable and accurate regression model. Metrics are orthogonalized into principal components and maintainability was modeled as a function of those components.

For initial construction and testing of the maintainability assessment models, eight software systems (ranging in size from 1,000 to 10,000 lines of code) were obtained from Hewlett-Packard (HP) along with subjective engineering assessments of the quality and maintainability of each code set. The survey used to gather the subjective data was constructed by systematically taking questions from the U.S. Air Force software maintainability evaluation instrument [1]. The AFOTEC instrument is intended to assess the maintainability of a software system from the source code and documentation of that system. It consists of a set of 171 statements which a group of surveyors must respond to on a forced-choice six point scale (ranging from completely disagree to completely agree). Each surveyor is asked to examine a set of software modules and respond to each statement in the instrument.

The surveys used in our studies were subsets of the AFOTEC instrument, which were completed by the HP engineers in charge of maintaining the test software systems. They were not aware of the intent behind the survey and were thus "blind" to our study. Responses to the surveys were tallied into a numerical rating that represented the maintainability of each system. For each of the eight software systems, a set of 40 software metrics were calculated for each module within the system. These metrics served as indicators of the complexity and/or quality inherent in each of the code samples. Using this data and the results from the expert's subjective assessment we constructed the seven maintainability models presented here. Table 4 summarizes the results and compares the maintainability indices obtained from the models against the HP software engineer's subjective maintainability rating. Statistical correlations between the calculated indices and the engineers' assessments are also shown in the table. Pearson Product-Moment correlation coefficients (r) indicate the correlation between the model and the HP rating. Spearman's Rank-Order correlation coefficients (ρ) show the the correlation between the rank ordering of the models' calculated indices and that obtained from Hewlett-Packard. As can be seen from the table, the correlations are excellent because the models were calibrated to the data.

In order to test the reliability of the models, six more HP software systems (ranging from 1,000 to 8,000 lines of code) were obtained and tested. The measured results were again compared with the HP software engineers' subjective maintainability scores, but this time the subjective maintainability scores

Table 4: Software Maintainability Model Results

Soft. Sys.	HP Rating	Hierarchy Assess model	Polynomial Regression 1 metric model	Polynomial Regression 4 metric model	Polynomial Regression 5 metric model	Entropy model	Factors model	Prin. Comp. model
A	61	16.62	63.6	61.3	60.0	69.60	56.00	58.89
B	66	76.71	74.8	75.1	71.4	72.66	82.06	73.47
C	69	70.59	74.8	68.3	65.8	73.81	82.89	79.41
D	71	78.31	73.3	69.1	74.4	73.46	80.89	72.77
E	81	80.02	76.6	76.1	77.5	76.48	84.69	75.04
F	92	89.84	79.8	88.8	90.7	77.89	83.03	81.46
G	94	95.95	84.2	96.4	94.2	76.61	88.75	90.29
H	95	97.67	78.5	93.9	95.0	76.31	81.69	97.66
		$\rho = .98^*$ $r = .78$	$\rho = .86^*$ $r = .85^*$	$\rho = .90^*$ $r = .95^*$	$\rho = .98^*$ $r = .97^*$	$\rho = .81$ $r = .90^*$	$\rho = .50$ $r = .60$	$\rho = .88^*$ $r = .86^*$

*Statistically significant ($\rho < 0.01$)

Table 5: Reliability Test Results

Soft. Sys.	HP Rating	Hierarchy Assess model	Polynomial Regression 1 metric model	Polynomial Regression 4 metric model	Polynomial Regression 5 metric model	Entropy model	Factors model	Prin. Comp. model
I	45.9	32.24	68.6	51.2	54.3	†	†	65.39
J	45.9	45.47	72.8	65.6	67.0	†	†	68.38
K	67.4	‡	70.8	63.8	71.0	†	†	67.67
L	86.6	82.66	81.6	76.7	81.2	†	†	82.84
M	76.8	91.22	80.5	86.0	91.8	†	†	82.05
N	76.8	91.99	82.0	88.2	92.4	†	†	79.52
		$\rho = .70$ $r = .93^*$	$\rho = .89^*$ $r = .85^*$	$\rho = .77$ $r = .78$	$\rho = .83^*$ $r = .83^*$			$\rho = .89^*$ $r = .88^*$

* Statistically Significant ($\rho < 0.05$)

† Not computable due to limited sample size.

‡ Not computable due to embedded C++ code.

were gathered using a different subset of the AFOTEC instrument containing questions which were systematically taken from each section of that instrument (details can be found in [22]). The purpose of this change was to explore the tolerance of the calibrated assessment models to the manual AFOTEC instrument.

Table 5 shows the reliability test result. The Hierarchical Multidimensional Assessment model, Polynomial Regression models, and Principal Components Regression model have shown high potential application in predicting software maintainability. Although the data are incomplete on the two complexity models, the theories behind the models are very attractive. More data needs to be gathered for further study and testing of all models.

4 Conclusion

In industry today, most assessments of software maintainability are presently carried out manually by experienced software engineering managers. Our work was designed to explore ways in which software maintainability may be more accurately and efficiently calculated using automated techniques.

Our Hierarchical Multidimensional Assessment model structure may be easily extended to assess far more than just the source program characteristics. The Principal Components Regression model is more accurate, but more complicated, than the Polynomial Regression Models. But the Polynomial Regression Models are useful to provide a quick, simple assessment of maintainability. The Principal Components model can be used when automated statistical techniques are available.

This study focused on source code as an independent variable to assess software maintainability. Future studies in software maintainability will include assessment of multidimensional characteristics which may influence software maintenance – such as operational environment, personnel management and supporting documentation – to get a more accurate software maintainability index.

References

[1] *Software Maintainability - Evaluation Guide*, an updated AFOTEC Pamphlet 800-2, vol. 3, Oct. 31, 1989, HQ Air Force Operational Test & Evaluation Center, Kirtland Air Force Base, NM 87117.

[2] G. Berns, "Assessing Software Maintainability," *Communications of the ACM*, Vol. 27(1), Jan. 1984, pp. 14-23.

[3] BDM Corp., *Software Maintainability Evaluator Guidelines Handbook*, report #BDM/TAC-78-687-TR, compiled for the Air Force Operational Test and Evaluation Center (Kirtland Air Force Base, NM) by the BDM Corp., 1801 Randolph Road SE, Albuquerque, NM 87106, 1978.

[4] BDM Corp., *Analysis of Software Maintainability Evaluation Process*, Report #BDM/TAC-78-698-TR, compiled for the Air Force Test and Evaluation Center (Kirtland Air Force Base, NM) by the BDM Corp., 1801 Randolph Road SE, Albuquerque, NM 87106, 1978.

[5] J. Collofello & M. Orn, "Improving Software Maintenance Skills in an Industrial Environment," *SEI Conference - 1989*, Springer-Verlag, New York, NY, 1989, pp. 26-36.

[6] V. Gibson & J. Senn, "System Structure and Software Maintenance Performance," *Communication of the ACM*, Vol. 32(3), Mar. 1989, pp. 347-358.

[7] L. Gremillion, "Determinants of Program Repair Maintenance Requirements," *Communications of the ACM*, Vol. 27(8), Aug. 1984, pp. 826-832.

[8] W. Harrison & C. Cook, "Insights on Improving the Maintenance Process Through Software Measurement," *Proceedings Conference on Software Maintenance - 1990*, (San Diego, CA, Nov. 26-29), IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 37-45.

[9] D. Kafura & G. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering* , Vol. 13(3), Mar. 1987, pp. 335-343.

[10] T. Korson & V. Vaishnavi, "An Empirical Study of the Effects of Modularity on Program Modifiability," *Empirical Studies of Programmers*, (Washington DC, June 5-6), Ablex Publishing Corp., Norwood, NJ, 1986, pp. 168-186.

[11] B. Lientz, E. Swanson, & G. Tompkins, "Characteristics of Application Software Maintenance," *Communications of the ACM*, Vol. 21(6), June 1978, pp. 466-471.

[12] J. Martin & W. Osborne, *Guidance on Software Maintenance*, US Department of Commerce - National Bureau of Standards, NBS Special Publication 500-106, US Government Printing Office, Washington DC, 1983.

[13] J. McCall & M. Matsumoto, *Software Quality Measurement manual*, prepared for the USAF Rome Air Development Center, by General Electric Company, Information Systems Programs, 450 Persian Dr., Sunnyvale, CA 94086, 1979.

[14] J. McCall, M. Herndon, & W. Osborne, *Software Maintenance Management*, US Department of Commerce - National Bureau of Standards, NBS Special Publication 500-129, US Government Printing Office, Washington DC, 1985.

[15] J. Munson & T. Khoshgoftaar, "Applications of a Relative Complexity Metric for Software Project Management," *Journal of Systems Software*, Dec. 1990, pp. 283-291.

[16] J. Munson & T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, Vol. 18(5), May 1992, pp. 423-433.

[17] T. Khoshgoftaar & J. Munson, "An Aggregate Measure of Program Module Complexity," *Proceedings Fourth Annual Oregon Workshop on Software Metrics*, March 22-24, 1992, Silver Falls, Oregon.

[18] P. Oman & C. Cook, "A Programming Style Taxonomy," *Proceedings CSC '90,* (18th. Annual ACM Computer Science Conference, Washington D.C), Feb. 1990, pp. 244-250.

[19] P. Oman, J. Hagemeister, & D. Ash, *A Definition and Taxonomy for Software Maintainability,* U.I. Software Engineering Test Lab Report #91-08 TR, Nov. 1991, 30 pages.

[20] P. Oman & J. Hagemeister, "Metrics for Assessing Software System Maintainability," *Proceedings of the 1992 IEEE Conference on Software Maintenance*, (Orlando FL, Nov. 9-12), IEEE Computer Society Press, Los Alamitos CA, 1992, pp. 337-344.

[21] P. Oman & J. Hagemeister, "Construction and Validation of Polynomials for Predicting Software Maintainability," to appear in *Proceedings of the Fifth Annual Oregon Workshop on Software Metrics*, March 21-23, 1993, Silver Falls, Oregon.

[22] P. Oman, *HP-MAS: A Tool for Software Maintainability Assessment*, U.I. Software Engineering Test Lab Report #92-07-ST, August 1992, 36 pages.

[23] D. Peercy, "A Software Maintainability Evaluation Methodology," *IEEE Transactions of Software Engineering* , Vol. 7(4), July 1981, pp. 144-152.

[24] M. Rees, "Automatic Assessment Aids for Pascal Programs," *ACM Sigplan Notices*, Vol. 17(10), Oct. 1982, pp. 33-42.

[25] H.Rombach, "Impact of Software Structure on Maintenance," *Proceedings Conference on Software Maintenance - 1985,* (Washington DC, Nov. 11-13), IEEE Computer Society Press, Washington DC, 1985, pp. 152-160.

[26] H. Rombach, "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, Vol. 13(3), Mar. 1987, pp. 344-354.

[27] H. Rombach & B. Ulery, "Improving Software Maintenance Through Measurement", *Proceedings of the IEEE*, Vol. 77(4), Apr. 1989, pp. 581-595.

[28] N. Schneidewind, "The State of Software Maintenance," *IEEE Transaction on Software Engineering*, Vol. 13(3), Mar. 1987, pp. 303-310.

[29] H. Sneed & A. Merey, "Automated Software Quality Assurance," *IEEE Transactions on Software Engineering*, Vol. 11(9), Sept. 1985, pp. 909-916.

[30] M. Van Emden, "An Analysis of Complexity," *Mathematical Centre Tracts*, 1971.

[31] I. Vessey & R. Weber, "Some Factors Affecting Program Repair Maintenance: An Empirical Study," *Communications of the ACM*, Vol.26(2), Feb. 1983, pp. 128-134.

[32] S. Yau & J. Collofello, "Design Stability Measures for Software Maintenance," *Proceedings of the 6th Annual International Conference on Software and Applications*, IEEE Computer Society, New York, NY, 1982, pp. 100-108.