Ordering Fault-Prone Software Modules

Taghi M. Khoshgoftaar*
Edward B. Allen
Florida Atlantic University
Boca Raton, Florida USA

Keywords — **software reliability, fault-prone modules, software quality models, module-order model, multiple linear regression**

Summary & Conclusions — **Software developers apply various techniques early in development to improve software reliability, such as extra reviews, additional testing, and strategic assignment of personnel. Due to limited resources and time, it is often not practical to enhance the reliability of all modules. Our goal is to target reliability enhancement activities to those modules that would otherwise have problems later. Prior research has shown that a software quality model based on software product and process metrics can predict which modules are likely to have faults.**

A *module-order model* is a quantitative software quality model that is used to predict the rank-order of modules according to a quality factor, such as the number of faults. The contribution of this paper is definition of module-order models and a method for their evaluation and use. Two empirical case studies of full-scale industrial software systems provide empirical evidence of the usefulness of module-order models for targeting reliability enhancement. Multiple linear regression models were the underlying quantitative models of the case studies; other techniques could also be used.

The subject of one case study was a military communications system. The subject of the other was a large legacy telecommunications system. The case studies demonstrated that even though an underlying quantitative model may not accurately predict the quality factor, using such a model for ordering modules can successfully target reliability enhancement efforts.

*Readers may contact the authors through Taghi M. Khoshgoftaar, Empirical Software Engineering Laboratory, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA. Phone: (561)297-3994, Fax: (561)297-2800, Email: taghi@cse.fau.edu, URL: www.cse.fau.edu/esel.html.

1 Introduction

A significant portion of software development cost is due to rework fixing defects. The amount of cost depends on when they are discovered. Correcting software faults late in the development life cycle is often very expensive. Consequently, software developers apply various techniques to discover faults early in development [7, 23]. Reliability enhancement techniques include more rigorous design and code reviews, automatic test-case generation to support more extensive testing, and strategic assignment of key personnel.

Our goal is to target reliability enhancement activities to those modules that are most likely to have problems. The goal is to enhance the reliability of modules recommended by a model earlier enough to prevent problems from poor reliability later in the life cycle. Software reliability can be directly measured only late in the life cycle, but product and process metrics can be measured substantially earlier. Prior research has shown that software product and process metrics [5] collected early in the software development life cycle can be the basis for reliability predictions using software quality models [9, 16, 20, 22, 27, 31, 33, 36]. Predicting the exact value of a reliability measure for each module is often not necessary; previous research has focused on classification models to identify *fault-prone* and *not fault-prone* modules [1, 4, 8, 13, 25]. Such models require that *fault-prone* be defined as a class before modeling, usually via a threshold on the number of faults expected. However, due to uncertain resource constraints that limit the amount of reliability enhancement effort, software development managers often cannot choose an appropriate threshold at the time of modeling. In such cases, a prediction of the rank-order of modules, from the least to the most *fault-prone*, is more useful [18]. In this paper, we focus on the degree that a module is *fault-prone*, rather than membership

in a class.

A *module-order model* has an underlying quantitative software quality model that is used to predict the rank-order of modules according to a quality factor, such as a measure of reliability. With a predicted rank-order in hand, one can select as many modules from the top of the list for reliability enhancement as resources will allow.

A module-order model has the characteristic that it can be used for classification [10]; the modules above a selected cutoff rank are like those classified as *fault-prone*. When actual fault data is known and a threshold defining *fault-prone* is given, one can determine whether each module was actually *fault-prone* or not, and thus, evaluate the classification accuracy of the model. In contrast to conventional classification modeling, one need not select a threshold that defines a *fault-prone* class prior to building a module-order model.

The contribution of this paper is definition of *module-order models* and a method for their evaluation and use. We also present two empirical case studies of full-scale industrial software systems that illustrate the use of module-order models for targeting reliability enhancement. Similar results from the two case studies are evidence that the technique is robust, in spite of differing organizations, programming languages, software metrics, and life cycle phases.

To be credible, the software engineering community demands that the subject of an empirical study be a system with the following characteristics [35]: (1) developed by a group, rather than an individual programmer; (2) developed by professionals, rather than students; (3) developed in an industrial environment, rather than an artificial setting; and (4) large enough to be comparable to real industry projects. The two case studies presented here fulfill these criteria. Our acronyms for the systems studied are below, followed by related software engineering nomenclature.

Acronyms

CCCS	Command, Control, and Communications System
LTS	Legacy Telecommunications System

Nomenclature

module	the unit of software modeled as an observation
Alberg diagram	a Pareto diagram for software reliability
code churn	lines of code added or changed
debug code churn	code churn due to fixing faults
development code churn	code churn due to development of new features
fit data set	data used to estimate parameters of a model
test data set	data used to evaluate model accuracy
fault	a defect in a software module (a "bug")
fault-prone	a set of modules that is likely to have many faults
software product metric	a measure of an attribute of a software product
software process metric	a measure of an attribute of a module's development process history
software quality factor	a measure of software quality
software quality model	a model that predicts a software quality factor
domain metric	a principal component when the underlying variables are software metrics

In related work, Ohlsson and Alberg introduce Pareto diagrams for software reliability which they call *Alberg* diagrams [28]. Similarly, Ohlsson, Helander and Wohlin use Alberg diagrams and Spearman rank-order correlation to evaluate models [29]. Curves are plotted for an ordering based on the actual number of faults, and an ordering based on the number of faults predicted by a model. When model predictions are intended for only ordinal purposes, Ohlsson et al. consider the model with the smallest area between curves to be the most useful. In case studies, they also compared models by the distance between the curves at selected percentages of modules. We have employed Alberg diagrams in this paper as an informal depiction of model accuracy.

Future work will investigate additional modeling techniques underlying the module-order model concept, such as genetic programming [15] and Poisson regression [9]. Future work will also compare the usefulness of module-order models and classification models [10].

2 Module-Order Modeling

According to Pareto's Law applied to software engineering, 20% of the modules will typically account for about 80% of the faults. This was approximately true in our case studies. Therefore, identifying the top fraction of the *fault-prone* modules can maximize the benefit of expending limited resources for reliability enhancement. Our goal is to develop models that will accurately predict the relative reliability of each module, especially those which are most *fault-prone*. A suitable software quality model can make predictions when it is not too late to take compensatory actions. Such predictions will be useful for prioritizing reliability enhancement efforts toward those modules offering the greatest payoff.

We define a module-order model as a software quality model, based on software product and process metrics, that is used to predict the rank-order of modules according to a quality factor. Most quality factors are directly measurable only after software has become operational. For example, the number of faults is known only after testing or operations. In contrast, software product and process metrics can be measured during development.

The input to a module-order model is a set of modules with measured product attributes and process history attributes. A module-order model consists of an underlying

quantitative model that predicts a quality factor for each module in the set followed by the step of ordering the modules by the predicted values. It is this ordering step that distinguishes a module-order model, irrespective of the underlying modeling technique or the specific quality factor. The output of a module-order model is a ranking of the modules in the set, rather than predicted quality factor values.

We are interested in quality factors that indicate software reliability. The predicted quality factor must have at least an ordinal scale [2]. This stands in contrast to classification models where the dependent variable is nominal scale, such as whether a module is *fault-prone* or not. For example, the quality factor could be the expected number of faults. Even though the number of faults is absolute scale when directly measured, our recommended use of a predicted value is only ordinal scale. Moreover, our approach is compatible with independent variables and modeling techniques which are ordinal scale or higher. Our case studies define software reliability in terms of the number of faults, or in terms of debug code churn. In the following discussion, we use the *number of faults* as an example quality factor. The same method applies to other quality factors, such as debug code churn.

In general, software quality modeling involves the following steps [32].

1. Build a model using historical data.

2. Evaluate the model using independent historical data.

3. Use the model on a current project's data.

Notation

n	number of modules
i	module identifier, $i = 1, \ldots, n$
$\mathbf{x}_i$	vector of independent variable values for module i
x_{ij}	j^{th} independent variable value for module i
F_i	quality factor for module i
F_{tot}	total of quality factor values, $F_{tot} \equiv \sum_{i=1}^{n} F_i$
$\widehat{F}(\mathbf{x}_i)$	estimate of F_i by model
$\mathbf{R}$	ranking of modules by F_i
R_i	percentile rank of module i in $\mathbf{R}$
$\widehat{\mathbf{R}}$	ranking of modules by $\widehat{F}(\mathbf{x}_i)$
$\widehat{R}(\mathbf{x}_i)$	percentile rank of module i in $\widehat{\mathbf{R}}$
c	cutoff percentile of ranked modules
$\mathcal{C}$	management's preferred set of cutoff percentiles
$G(c)$	quality of modules above c percentile under perfect ranking
$\widehat{G}(c)$	quality of modules above c percentile under predicted ranking
$\phi(c)$	performance measure for module-order models
$\mathbf{Z}$	$n \times m$ matrix of standardized measurements
Z_j	standardized variables $j = 1, \ldots, m$
λ_j	j^{th} eigenvalue of the covariance matrix of $\mathbf{Z}$
$\mathbf{e}_j$	j^{th} eigenvector of the covariance matrix of $\mathbf{Z}$
$\mathbf{T}$	standardized transformation matrix, $n \times p$
$\mathbf{t}_j$	j^{th} column of $\mathbf{T}$
D_j	j^{th} domain metric
α	significance level for hypothesis test
y_i	dependent variable value for module i
$\hat{y}_i$	estimate of y_i
a_j	parameter for j^{th} independent variable
e_i	error term for observation i
R^2	coefficient of determination
AAE	average absolute error
ARE	average relative error

See Tables 1 and 4 for software metrics notation. Other standard notation is given in "Information for Readers and Authors" in the rear of each issue.

The underlying quantitative model is built according to the applicable modeling technique. For example, a parametric model is an equation where a software quality factor, $\widehat{F}(\mathbf{x}_i)$, is a function of independent variables, $x_{i1}, \ldots, x_{ip}$ having a certain functional form and parameters that need to be estimated.

$$\widehat{F}(\mathbf{x}_i) = f(x_{i1}, \ldots, x_{ip}) \tag{1}$$

A modeling technique includes methods for selecting independent variables and for estimating parameters.

Because we do not expect perfect accuracy, we evaluate a model's usefulness by its ability to approximately order modules from the most *fault-prone* to a certain percentile. We propose the following method for evaluating a module-order model.

1. Determine the perfect ranking of modules in the *test* data set, $\mathbf{R}$, by ordering modules according to F_i.

2. Determine the predicted ranking, $\widehat{\mathbf{R}}$, by ordering modules in the *test* data set according to $\widehat{F}(\mathbf{x}_i)$ from least to most *fault-prone*.

3. When applying a module-order model, modules are selected for reliability enhancement in predicted order, beginning with the most *fault-prone*. A *cutoff* point, c, is the percentile of the last module enhanced.

   For evaluation of a model, we choose a set of cutoff points, $\mathcal{C}$, that might be of interest to a manager. In the case studies, we chose 50 through 95 percentile, in 5 percent increments. Another project might choose different percentiles, but this set illustrates our methodology.

   For each cutoff percentile value of interest, $c \in \mathcal{C}$:

(a) Calculate the sum of actual number of faults, $G(c)$, in modules above the cutoff for $\mathbf{R}$.

$$G(c) \equiv \sum_{i:R_i \geq c} F_i \tag{2}$$

(b) Calculate the sum of actual number of faults in modules above the cutoff, $\widehat{G}(c)$, for $\widehat{\mathbf{R}}$.

$$\widehat{G}(c) \equiv \sum_{i:\widehat{R}(\mathbf{x}_i) \geq c} F_i \tag{3}$$

4. Calculate the percentage of faults accounted for by each ranking, namely, $G(c)/F_{tot}$ and $\widehat{G}(c)/F_{tot}$. Evaluate the benefit of using the model at selected values of c

5. Calculate a measure of model performance that indicates how closely the faults accounted for by the model ranking match those of the perfect ranking.

$$\phi(c) \equiv \frac{\widehat{G}(c)}{G(c)} \tag{4}$$

The proportion of the actual faults at c, $\phi(c)$, indicates the accuracy of the model's ranking at that c compared to a perfect ranking. In the face of uncertain resources for reliability enhancement, we are interested in consistent accuracy over a range of c. The variation in $\phi(c)$ over a range of c indicates the robustness of the model; small variation implies a robust model.

In practice, a manager is interested in the model's accuracy only at the preferred cutoff point, c. Because all modules above the cutoff point get the same treatment, the distance of the predicted rank-order from the actual is not an appropriate measure of model accuracy. We do not care about the accuracy of the rank-order within the enhanced group, nor within the group that was not enhanced. However, we do want $\phi(c)$ to be high for the c of interest.

Because the cutoff point may not be known in advance, a manager may also be interested in the accuracy of the model over a range of cutoff points represented by the set $\mathcal{C}$. If the accuracy is consistent over a range of interest, we consider the model *robust*. Because the set $\mathcal{C}$ has few members, a graphical presentation of accuracy and robustness is preferred instead of summary statistics.

Spearman rank-order correlation represents overall accuracy over the entire data set; it is not an appropriate measure of accuracy here because a manager is only interested in the accuracy of the model at his preferred value of c. Spearman correlation is also not a measure of robustness, because it does not measure variation in accuracy. The area between curves of an Alberg diagram [28] is also a measure of overall accuracy, and similarly, it is not appropriate for evaluation of module-order models, because it represents accuracy over the entire data set.

After evaluating the accuracy and robustness of a module-order model, it is ready for use on a current similar project, or subsequent release. Determine the predicted ranking, $\widehat{\mathbf{R}}$, by ordering modules in the current data set according to $\widehat{F}(\mathbf{x}_i)$.

3 Case Study Methodology

Because software development is inherently a people-intensive enterprise, software reliability is influenced by many factors that vary tremendously among organizations. Consequently, credible controlled experiments to validate an analysis method are not practical [30]. Therefore, we take a case-study approach to illustrate the usefulness of module-order models in a real-world setting. Case studies provide “weight of evidence”, rather than scientific proof of propositions.

A case study is based on data from one or more past development projects. The case study builds software quality models that could have been built at an early point in the life cycle and calculates predictions of the rank-order. The predictions are compared to the rank-order based on the actual reliability experienced in a later phase of the life cycle. Thus, a case study simulates what could have been predicted during the development project, and evaluates the result.

The case study results indicate the accuracy that one can expect from the model when applied to a similar current project. The following is a summary of our case study methodology which is applicable to a wide variety of modeling techniques.

1. Preprocess measurements, if necessary.

   If incompatible units of measure complicate interpretation of the model, standardize measurements to a mean of zero and a variance of one for each metric, so that the unit of measure becomes one standard deviation. If multicollinearity could degrade the stability of the model, perform principal components analysis on the standardized product metrics to produce domain metrics.

2. Choose a model validation strategy. A validation strategy evaluates the accuracy of a model by estimating model parameters with a *fit* data set, predicting the dependent variable of each observation in the *test* data set, and then evaluating the results. However, each of the following methods defines the *test* data set differently [3].

   (a) *Resubstitution.* This method uses the *fit* data set as a *test* data set. Since the *fit* and *test* data sets are not at all independent, this is the least realistic of

the methods discussed here. This assessment of model accuracy is often overly optimistic.

(b) *Data splitting.* This is sometimes called the "Holdout" method [26]. *Fit* and *test* data sets are derived from a single data set by impartially partitioning available observations. Statistical similarity of the *fit* and *test* data set is assured by the partitioning method. Data splitting may be appropriate when data on a similar subsequent project is not available. The proportion of observations in each data set is chosen according to sample sizes needed for the statistical techniques to be employed. Consequently, this method often requires large samples of data. Our first case study used this method.

(c) *Cross-validation.* This is sometimes called the "U-Method" [26]. Suppose there are n observations available. Let one observation be the current *test* data set and all the others be the current *fit* data set. Build a model with the current *fit* data set, and evaluate it for the current *test* data set. Repeat for each observation, resulting in n models. Let the results summarize the n evaluations of the models. In contrast to resubstitution, this does not have serious bias. This method is appropriate for smaller data sets than data splitting, but involves more computation.

(d) *Subsequent project.* This method uses data from one past project as a *fit* data set and data from a similar past project or subsequent release as a *test* data set. This is a realistic simulation of applying a software quality model in practice. Our second case study used this method.

3. Prepare *fit* and *test* data sets according to the model validation strategy.

A quality factor is the dependent variable and software product metrics, process metrics, and/or their transformations are the independent variables.

4. Select significant independent variables from a set of candidates, based on the *fit* data set.

5. Estimate parameters of the model on the selected independent variables, based on the *fit* data set.

6. Evaluate the accuracy of the model using the *test* data set and the method presented in Section 2. The results indicate the accuracy and robustness that one can expect when applying the model to a similar project or subsequent release.

Each case study of this paper used a simple quantitative modeling technique, multiple linear regression, to predict a measure of software reliability. The simple models adequately fulfill the purposes of this paper, without resorting to refinements of multiple linear regression, such as identification of outliers. Other quantitative modeling techniques could be used, such as nonlinear regression [14], regression trees [6], or computational intelligence techniques [1, 19, 21]. The following is a summary of the technique used here.

The first step is to select independent variables. Even though we may have a long list of candidate independent variables, it is possible that some do not significantly influence the dependent variable. If an insignificant variable is included in the model, it may add noise to the results and may cloud interpretation of the model. In particular, if a coefficient, a_j, for the jth variable in a linear model is not significantly different from zero, then it is best to omit that term from the model. The process of determining which variables

are significant is called *model selection*. Of several model selection techniques available for multiple linear regression, we use the *stepwise regression* method [26]. Stepwise model selection is an iterative procedure. All the candidate independent variables are specified. Significant variables are added and insignificant variables are removed from the model on each iteration, based on an F test. The test is recomputed on each iteration, until no variable can be added to or removed from the model.

Many models have a general mathematical form with parameters that must be chosen so that the *fit* data set matches the model as closely as possible. The second step consists of estimating the values of such parameters. In general, a multivariate linear model has the following form.

$$\hat{y}_i = a_0 + a_1 x_{i1} + \ldots + a_p x_{ip} \quad (5)$$

$$y_i = a_0 + a_1 x_{i1} + \ldots + a_p x_{ip} + e_i \quad (6)$$

where $e_i = y_i - \hat{y}_i$ is the error for the ith observation. We estimate the parameters, $a_0, \ldots, a_p$, using the *least squares* method. This method chooses a set of parameter values that minimizes $\sum_{i=1}^{n} e_i^2$ [26]. Other estimation techniques could be used [17].

When the parameters have been estimated, and given a set of independent variable values, a model can calculate a value of the dependent variable. Since the independent variables are known earlier than the actual value of the dependent variable, the calculated value is a *prediction*.

When we know the actual dependent variable value, y_i, for the ith module, we validate that the predictions, $\hat{y}_i$, are sufficiently accurate for the needs of the current project. Two common statistics for evaluating quantitative predictions are average absolute error,

AAE, and average relative error, ARE.

$$AAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \quad (7)$$

$$ARE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i + 1}\right| \quad (8)$$

where the denominator of ARE has one added to avoid dividing by zero [17]. A lower average error is better. Results based on the *fit* data set indicate quality of fit. Results based on the *test* data set indicate the accuracy of predictions one can expect for a similar current project or a subsequent release.

4 A Military Communications System

4.1 System Description

We studied a large military system written in Ada for command, control, and communications which we call "CCCS". Generally, a module was an Ada package, consisting of one or more procedures. A problem reporting system collected data on faults during the system integration and test phase and during the first year of deployment. Each fault was attributed to a module. The top 20% of the modules contained 82.2% of the faults. 52% of the modules had no faults, and over three quarters of the modules had two or fewer faults. The maximum number of faults in one module was 42. The developers collected software product metrics from the source code of each module. The number and selection of metrics was determined by available data collection tools. Table 1 lists the software product metrics used in this study. Another project might collect a different set [12].

Table 1: Software Product Metrics for CCCS

Symbol	Description
η_1	Number of unique operators
N_1	Total number of operators
η_2	Number of unique operands
N_2	Total number of operands
$V(G)$	McCabe's cyclomatic complexity
$V'(G)$	Extended cyclomatic complexity
	$V'(G) = V(G) +$ number of logical operators
LOC	Lines of code
$ELOC$	Executable lines of code

We used the 282 modules measured by the developers for our experiment. Applying data splitting, we impartially partitioned this data into two subsets, two thirds of the modules (188) for fitting models, and the remaining third (94 modules) for validating their predictive accuracy. This yielded adequate sample sizes for statistical purposes.

4.2 Model

We applied multiple linear regression to CCCS data. The predicted dependent variable, $\widehat{F}(\mathbf{x}_i)$, was the number of faults. Candidate independent variables, $\mathbf{x}_i$, were eight product metrics. Based on the *fit* data set, stepwise regression selected η_2, $V(G)$, $V'(G)$, and η_1 at the 5% significance level. The intercept was not significantly different from zero. The following model was estimated using the least squares technique.

$$\widehat{F} = 0.0306\,\eta_2 - 0.2632\,V(G) + 0.2314\,V'(G) - 0.0425\,\eta_1 \quad (9)$$

Each variable was significant at $\alpha < 0.04$. The metrics $V(G)$ and $V'(G)$ together indicate the amount of program logic. The numbers of distinct operators (η_1) and distinct

operands (η_2) are correlated with overall size of the module. Thus, modules with intricate program logic and large size are more likely to have faults.

The quality of fit was indicated by an $R^2 = 0.738$, and by resubstitution of the *fit* data set, which yielded average absolute error of $AAE = 1.5$ faults, and average relative error of $ARE = 0.62$. Application of the model to the *test* data set yielded $AAE = 2.2$ faults and $ARE = 0.70$.

Evaluating this model as a conventional quantitative model, the quality of fit was satisfactory. AAE and ARE for the *test* data set indicated that the model was accurate enough to be useful to the project. Even so, we did not use this model in a conventional manner. We used it only to order the modules.

4.3 Evaluation

We applied the evaluation method presented in Section 2. Table 2 lists results for the module-order model of CCCS. Since over half of the modules had no faults, reliability enhancement of more than half of the modules would have had minimal payoff here. Therefore, we did not analyze the model for the first and second quartiles, $c < 0.50$.

Suppose the project manager planned to enhance the reliability of modules in the order recommended by the model. When done, software engineers would have found that enhanced modules contained a certain fraction of the faults. Figure 1 depicts this with an Alberg diagram [28] for a perfect ordering of modules ($G(c)/F_{tot}$ in Table 2) and for the CCCS model ($\widehat{G}(c)/F_{tot}$ in Table 2). The cumulative percentage of faults in modules is plotted as a function of the percentage of modules recommended $(1 - c)$, where modules are ordered from most *fault-prone* to least. Figure 2 shows the ratio of the two lines in

Table 2: Results of CCCS Model

c	$G(c)/F_{tot}$	$\widehat{G}(c)/F_{tot}$	$\phi(c)$
0.950	0.419	0.361	0.861
0.900	0.631	0.544	0.862
0.850	0.751	0.697	0.928
0.800	0.822	0.772	0.939
0.750	0.884	0.830	0.939
0.700	0.925	0.846	0.915
0.650	0.942	0.871	0.925
0.600	0.963	0.884	0.918
0.550	0.983	0.909	0.924
0.500	1.000	0.913	0.913

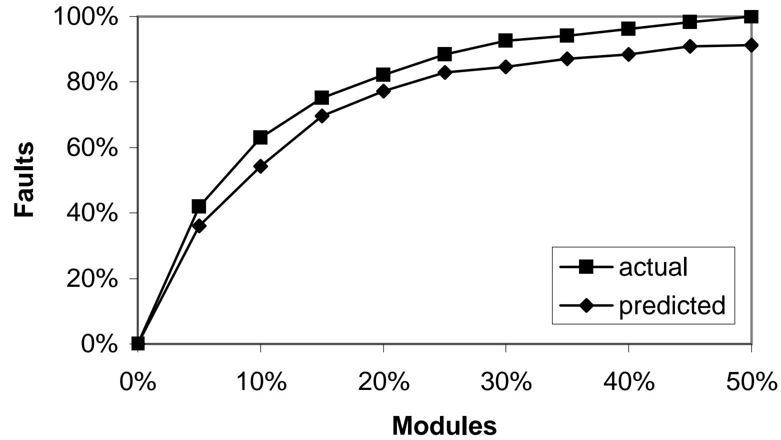

Figure 1: Alberg Diagram for CCCS Model

Figure 1, which is a measure of model performance ($\phi(c)$ in Table 2). In other words, it shows how close the model came to a perfect ordering of the modules for the most *fault-prone* half of the modules.

The most *fault-prone* 5% of the modules that the model recommended for reliability enhancement ($c = 0.95$) accounted for 36.1% of the faults. This corresponded to 86.1% of the faults that a perfect recommendation would account for. Even a modest investment

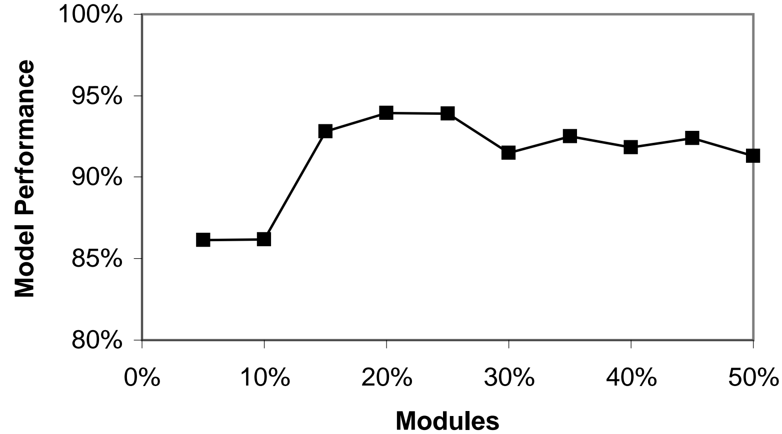

Figure 2: Performance of CCCS Model

in reliability enhancement, i.e., 5% of the modules, would yield early detection of many faults, i.e., up to 36.1%.

Figure 2 illustrates that model performance, $\phi(c)$, is about the same for the range of c of interest; we consider the model very robust. Model accuracy was not substantially affected by how many modules would be actually given reliability enhancement. We used a simple well-known quantitative modeling technique in this case study, namely, multiple linear regression, to show that a module-order model is useful and robust, even though the underlying quantitative model may be less than ideal.

5 A Legacy Telecommunication System

5.1 System Description

We studied a large legacy telecommunications system (LTS) written by professional programmers in a large organization [11]. This embedded computer application included numerous finite state machines and interfaces to other kinds of equipment. It was writ-

Table 3: Debug Code Churn Statistics for LTS

Lines new/changed due to bug fixes, *FIX_NC*

	Release	
Statistic	0	1
Obs	97	171
Mean	154.7	331.6
Std Dev	300.0	734.9
Min	0	0
Q1	11	12
Median	77	68
Q3	199	294
Max	2439	5066

ten in a proprietary procedural high level language similar to Pascal. The entire system (Release 1) had over 50 thousand procedures; the portion we studied had over 38 thousand procedures in 171 modules.

Prior research has shown that the reuse history of a module can be a significant input to software quality models [13]. This case study considered only those modules that were new or changed since the prior release. The unchanged modules were extremely reliable, i.e., all had no changes due to debugging. *Development code churn*, *DEV_NC*, quantified the amount of code that was not reused from the previous release. *DEV_NC* can be measured at the same point in the life cycle as software metrics.

In this study, we focused on *debug code churn*, *FIX_NC*. Bug fixes generally occur after development coding is complete. Table 3 gives summary statistics on debug code churn for both releases. This data was available at the module level.

Software product metrics were collected by a Logiscope® metric analyzer at the procedure level, and were aggregated to the module level. Metrics missing relevant data,

Table 4: Software Product Metrics for LTS

Symbol	Description
N_STMTS	Number of statements.
N_COM	Number of comments.
TOT_OPTR	Total operators. (i.e., N_1 in CCCS case study)
N_EDGES	Number of edges in control flow graph.
N_NODES	Number of nodes in control flow graph.
P_NODES	Number of pending nodes. The number of nodes that begin a sequence of unreachable code (dead code).
MAX_DEG	Maximum degree of node. The maximum number of edges going in or out of a node.
N_PATHS	Number of independent paths.
N_IN	Number of entries. Since each procedure has only one entry point, this is equivalent to the number of procedures in the module.
N_STRUC	Number of control structures. (e.g., `IF`, `WHILE`, or `DO`)
MAX_LVLS	Maximum nesting levels.
N_SEQ	Number of sequential nodes.
MAX_NODES	Maximum nodes in a control structure.
MAX_STMTS	Maximum statements in control structure.
DRCT_CALLS	Direct calls. This is the total number of procedure or function calls.
ESS_CPX	McCabe essential complexity. This measures the degree a module has constructs due to branching in/out of control structures. A well structured module has $ESS_CPX = 1$.
DES_CPX	McCabe design complexity [24], which is intended to measure the cyclomatic complexity related to procedure calls.

constant metrics, and metrics that are linear combinations of others were excluded. Many of the metrics are defined on a control flow graph of the module, consisting of nodes and directed edges. Table 4 lists the software product metrics used in this study.

Data for each module consisted of product metrics, development code churn, *DEV_NC*, and debug code churn, *FIX_NC*. New or changed modules from Release 0 were used as a *fit* data set (97 observations), and new or changed modules from Release 1 were used

as a *test* data set (171 observations).

5.2 Preprocess Measurements

The first case study took a very simple approach to quantitative modeling. In this case study, we added preprocessing of data to reduce the large number of product metrics to a few principal components.

Software product metrics have a variety of units of measure, which are not readily combined in a multivariate model. In this case study, we transformed all product metric variables, so that each standardized variable had a mean of zero and a variance of one. Thus, the common unit of measure became one standard deviation.

Principal components analysis is a statistical technique for transforming multivariate data into orthogonal variables, and for reducing the number of variables without losing significant variation. The following summarizes the technique [34].

Suppose we have m measurements on n modules. Let $\mathbf{Z}$ be the $n \times m$ matrix of standardized measurements where each row corresponds to a module and each column is a standardized variable. Our principal components are linear combinations of the m standardized random variables, $Z_1, \ldots, Z_m$. The principal components represent the same data in a new coordinate system, where the variability is maximized in each direction and the principal components are uncorrelated. If the covariance matrix of $\mathbf{Z}$ is a real symmetric matrix with distinct roots, then one can calculate eigenvalues, λ_j, and eigenvectors, $\mathbf{e}_j, j = 1, \ldots, m$, of the covariance matrix of $\mathbf{Z}$. Each eigenvalue is the variance of the corresponding principal component. Since the eigenvalues form a nonincreasing series, $\lambda_1 \geq \ldots \geq \lambda_m$, one can reduce the dimensionality of the data without significant

loss of explained variance by considering only the first p components, $p \ll m$, according to some stopping rule, such as achieving a threshold of explained variance. For example, choose the minimum p such that $\sum_{j=1}^{p} \lambda_j / m \geq 0.90$ to achieve at least 90% of explained variance.

Let $\mathbf{T}$ be the $m \times p$ standardized transformation matrix whose columns, $\mathbf{t}_j$, are defined as

$$\mathbf{t}_j = \frac{\mathbf{e}_j}{\sqrt{\lambda_j}} \text{ for } j = 1, \ldots, p \tag{10}$$

Let D_j be a principal component random variable, and let $\mathbf{D}$ be an $n \times p$ matrix with D_j values for each column, $j = 1, \ldots, p$.

$$D_j = \mathbf{Z}\mathbf{t}_j \tag{11}$$

$$\mathbf{D} = \mathbf{Z}\mathbf{T} \tag{12}$$

When the underlying data is software metric data, we call each D_j a *domain metric.*

Principal components analysis of the combined *fit* and *test* software product metrics of LTS retained five components under the stopping rule that we retain components accounting for at least 90% of the overall variance. We combined the data sets for this step to improve the sample size. Table 5 shows the correlation between each original metric and each domain metric. The largest in each row is shown bold, indicating the product metrics that dominate each domain metric. The principal components analysis resulted in a standardized transformation matrix which was separately applied to the *fit* and *test* standardized product metrics to calculate domain metrics, $D_1, \ldots, D_5$.

Table 5: Domain Pattern for LTS Product Metrics

	Domain Metric				
	D1	D2	D3	D4	D5
P_NODES	**0.933**	0.099	0.203	0.054	0.124
ESS_CPX	**0.881**	0.356	0.236	0.119	0.106
DES_CPX	**0.880**	0.353	0.238	0.112	0.113
N_STRUC	**0.813**	0.306	0.255	0.244	0.306
N_NODES	**0.752**	0.512	0.246	0.212	0.230
N_EDGES	**0.733**	0.508	0.261	0.226	0.250
N_STMTS	**0.699**	0.590	0.200	0.193	0.222
N_COM	**0.686**	0.533	0.188	0.169	0.201
TOT_OPTR	0.213	**0.921**	0.146	0.040	0.003
DRCT_CALLS	0.266	**0.918**	0.104	0.047	0.010
N_IN	0.460	**0.831**	0.136	0.102	-0.005
N_SEQ	0.566	**0.724**	0.231	0.189	0.168
MAX_DEG	0.118	**0.586**	0.569	-0.229	-0.184
MAX_NODES	0.298	0.154	**0.908**	0.125	0.126
MAX_STMTS	0.286	0.163	**0.887**	0.150	0.142
MAX_LVLS	0.272	0.091	0.130	**0.922**	0.057
N_PATHS	0.408	-0.017	0.145	0.060	**0.876**
Variance	6.224	4.821	2.506	1.248	1.210
% Variance	36.6%	28.4%	14.7%	7.3%	7.1%
Cumulative	36.6%	65.0%	79.7%	87.0%	94.1%

5.3 Model

We applied multiple linear regression to LTS data. The predicted dependent variable, $\hat{F}(\mathbf{x}_i)$, was debug code churn, *FIX_NC*. Candidate independent variables were *DEV_NC* and domain metrics D_1 through D_5. Based on the *fit* data set, stepwise regression selected *DEV_NC* and D_1 at the 5% significance level. The following model was estimated using the least squares technique.

$$\hat{F} = 95.654 + 0.289\, DEV_NC + 63.274\, D_1 \qquad (13)$$

Each variable was significant at $\alpha < 0.033$. The pattern of principal components in Table 5 revealed that D_1 was associated with overall module size. In particular, the raw metrics that were highly correlated with D_1 are generally associated with size. As discussed above, DEV_NC indicated the amount of new and changed code for the current release. Thus, large modules with extensive new or changed code are more likely to have faults.

The quality of fit was indicated by an $R^2 = 0.138$, and by resubstitution of the *fit* data set which yielded average absolute error of $AAE = 135.0$ lines of code, and average relative error of $ARE = 22.2$. Application of the model to the *test* data set yielded $AAE = 188.5$ lines of code and $ARE = 35.3$.

Evaluating this model as a conventional quantitative model, the model was significant, but quality of fit was low. AAE and ARE for the *test* data set indicated that the model was not very accurate. However, we did not use this model in a conventional manner. We used it only to order the modules.

5.4 Evaluation

We applied the evaluation method presented in Section 2. Table 6 lists results for the module-order model of LTS. Since the median debug code churn was substantially below the mean, reliability enhancement of more than half of the modules would likely not be worthwhile. Moreover, limited resources for reliability enhancement would make enhancement of more than half impractical. Therefore, we did not analyze the model for modules below the median, $c < 0.50$.

Suppose the project manager planned to enhance the reliability of modules in the

Table 6: Results of LTS Model

c	$G(c)/F_{tot}$	$\widehat{G}(c)/F_{tot}$	$\phi(c)$
0.950	0.434	0.374	0.861
0.900	0.639	0.568	0.889
0.850	0.738	0.684	0.928
0.800	0.817	0.757	0.927
0.750	0.864	0.791	0.915
0.700	0.905	0.834	0.922
0.650	0.929	0.858	0.924
0.600	0.948	0.910	0.960
0.550	0.962	0.922	0.959
0.500	0.974	0.940	0.965

order recommended by the model. When done, software engineers would have found that the enhanced fraction of modules would have contained a certain fraction of the debug code churn. Figure 3 depicts this with an Alberg diagram [28] for a perfect ordering of modules ($G(c)/F_{tot}$ in Table 6) and for the LTS model ($\widehat{G}(c)/F_{tot}$ in Table 6). The cumulative percentage of debug code churn in modules is plotted as a function of the percentage of modules recommended $(1-c)$, where modules are ordered from most *fault-prone* to least. Figure 4 shows model performance, i.e., the ratio of the two lines in Figure 3 ($\phi(c)$ in Table 6). In other words, it shows how close the model came to a perfect ordering of the modules for the most *fault-prone* half of the modules.

The most *fault-prone* 5% of the modules that the model recommended for reliability enhancement ($c = 0.95$) accounted for 37.4% of the debug code churn. This corresponded to 86.1% of the debug code churn that a perfect recommendation would account for. This means that even a modest investment in reliability enhancement, i.e., 5% of the modules, would yield early detection of many faults, i.e., up to 37.4% of the debug code churn.

Since model performance, $\phi(c)$, is about the same for the range of c of interest, we

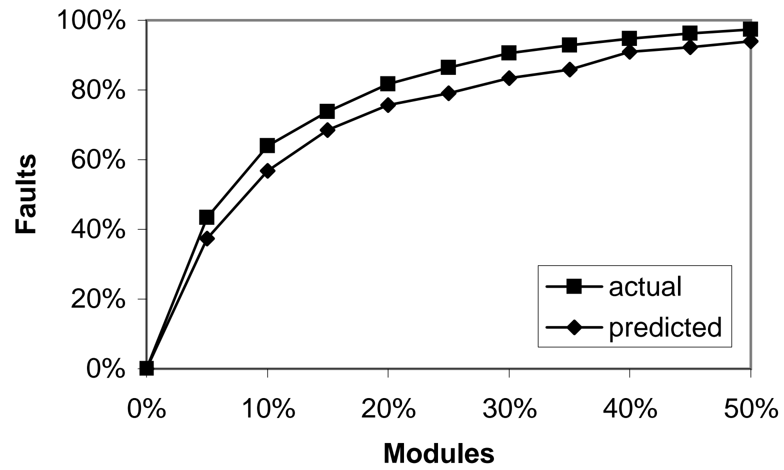

Figure 3: Alberg Diagram for LTS Model

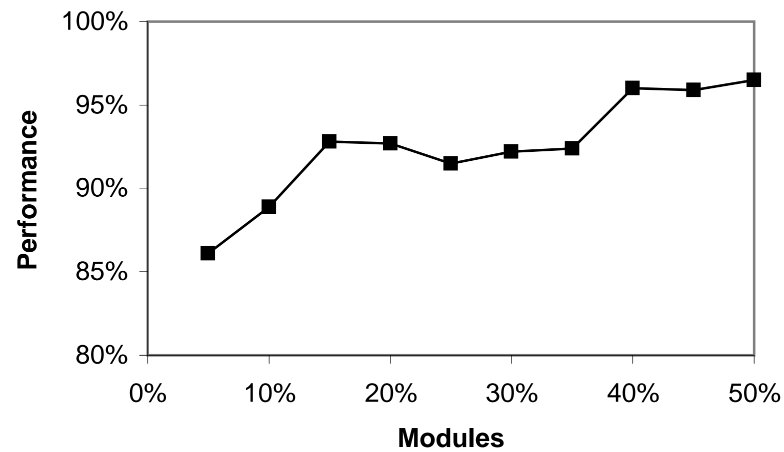

Figure 4: Performance of LTS Model

consider the model very robust. In other words, model accuracy did not depend on how many modules would be actually given reliability enhancement.

We used multiple linear regression in this case study, demonstrating that a module-order model can be useful and robust. Even though the underlying quantitative model was less accurate than the CCCS regression model, the module-order model results were about the same, providing evidence that this technique is robust, even though the projects were performed by different organizations in different programming languages, with different software metrics, and in different life cycle phases.

Acknowledgments

We thank Nishith Goel, Amit Nandi, and John McMullen for providing the data on the legacy telecommuniations system. We thank Jason Busboom for data analysis calculations. We thank anonymous reviewers for their thoughtful comments. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of collaborators. Moreover, our results do not in any way reflect the quality of our collaborators' software products. Logiscope® is a trademark of Verilog, S.A.

References

[1] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering*, 19(11):1028–1044, Nov. 1993.

[2] L. C. Briand, K. El Emam, and S. Morasca. On the application of measurement theory in software engineering. *Empirical Software Engineering: An International Journal*, 1(1):61–88, 1996.

[3] W. R. Dillon and M. Goldstein. *Multivariate Analysis: Methods and Applications.* John Wiley & Sons, New York, 1984.

[4] C. Ebert. Classification techniques for metric-based software development. *Software Quality Journal*, 5(4):255–272, Dec. 1996.

[5] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing, London, 2d edition, 1997.

[6] S. S. Gokhale and M. R. Lyu. Regression tree modeling for the prediction of software quality. In H. Pham, editor, *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*, pages 31–36, Anaheim, CA, Mar. 1997. International Society of Science and Applied Technologies.

[7] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. EMERALD: Software metrics and models on the desktop. *IEEE Software*, 13(5):56–60, Sept. 1996.

[8] T. M. Khoshgoftaar and E. B. Allen. The impact of costs of misclassification on software quality modeling. In *Proceedings of the Fourth International Software Metrics Symposium*, pages 54–62, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.

[9] T. M. Khoshgoftaar and E. B. Allen. An information theoretic approach to predicting software faults. *International Journal of Reliability, Quality and Safety Engineering*, 5, 1998. Forthcoming.

[10] T. M. Khoshgoftaar and E. B. Allen. Predicting the order of fault-prone modules in legacy software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 344–353, Paderborn, Germany, Nov. 1998. IEEE Computer Society.

[11] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 364–371, White Plains, NY, Oct. 1996. IEEE Computer Society.

[12] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. Flass. Process measures for predicting software quality. *Computer*, 31(4):66–72, Apr. 1998.

[13] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, Jan. 1996.

[14] T. M. Khoshgoftaar, B. B. Bhattacharyya, and G. D. Richardson. Predicting software errors during development using nonlinear regression models: A comparative study. *IEEE Transactions on Reliability*, 41(3):390–395, Sept. 1992.

[15] T. M. Khoshgoftaar, M. P. Evett, E. B. Allen, and P.-D. Chien. An application of genetic programming to software quality prediction. In W. Pedrycz and J. F. Peters, editors, *Computational Intelligence and Software Engineering*. World Scientific, Singapore, 1998. Forthcoming.

[16] T. M. Khoshgoftaar and J. C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, Feb. 1990.

[17] T. M. Khoshgoftaar, J. C. Munson, B. B. Bhattacharya, and G. D. Richardson. Predictive modeling techniques of software quality from software measures. *IEEE Transactions on Software Engineering*, 18(11):979–987, Nov. 1992.

[18] T. M. Khoshgoftaar, J. C. Munson, and D. L. Lanning. Alternative approaches for the use of metrics to order programs by complexity. *Journal of Systems and Software*, 24(3):211–221, Mar. 1994.

[19] T. M. Khoshgoftaar, A. S. Pandya, and H. B. More. A neural network approach for predicting software development faults. In *Proceedings of the Third International Symposium on Software Reliability Engineering*, pages 83–89, Research Triangle Park, NC USA, Oct. 1992. IEEE Computer Society.

[20] D. L. Lanning and T. M. Khoshgoftaar. The impact of software enhancement on software reliability. *IEEE Transactions on Reliability*, 44(4):677–682, Dec. 1995.

[21] D. B. Leake. CBR in context: The present and future. In D. B. Leake, editor, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, chapter 1, pages 3–30. MIT Press, Cambridge, MA USA, 1996.

[22] M. R. Lyu. Introduction. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 1, pages 3–25. McGraw-Hill, New York, 1996.

[23] M. R. Lyu, J. S. Yu, E. Keramidas, and S. R. Dalal. ARMOR: Analyzer for reducing module operational risk. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 137–142, Pasadena, CA, June 1995. IEEE Computer Society.

[24] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, Dec. 1989.

[25] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, May 1992.

[26] R. H. Myers. *Classical and Modern Regression with Applications*. Duxbury Series. PWS-KENT Publishing, Boston, 1990.

[27] A. P. Nikora, N. F. Schneidewind, J. C. Munson, and G. A. Hall. IV&V issues in achieving high reliability and safety in critical control system software. In H. Pham, editor, *Proceedings of the Third ISSAT International Conference on Reliability and*

Quality in Design, pages 25–30, Anaheim, CA, Mar. 1997. International Society of Science and Applied Technologies.

[28] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, Dec. 1996.

[29] N. Ohlsson, M. Helander, and C. Wohlin. Quality improvement by identification of fault-prone modules using software design metrics. In *Proceedings of the Sixth International Conference on Software Quality*, pages 2–13, Ottawa, Ontario, Canada, Oct. 1996. Sponsored by ASQC.

[30] S. L. Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering*, 1:219–253, 1995.

[31] S. L. Pfleeger. Assessing measurement. *IEEE Software*, 14(2):25–26, Mar. 1997. Editor's introduction to special issue.

[32] N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, May 1992.

[33] N. F. Schneidewind. Software metrics validation: Space Shuttle flight software example. *Annals of Software Engineering*, 1:287–309, 1995.

[34] G. A. F. Seber. *Multivariate Observations*. John Wiley and Sons, New York, 1984.

[35] L. G. Votta and A. A. Porter. Experimental software engineering: A report on the state of the art. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 277–279, Seattle, WA, Apr. 1995. IEEE Computer Society.

[36] M. A. Vouk and K. C. Tai. Some issues in multi-phase software reliability modeling. In *Proceedings of the CASCON*, pages 513–523, Toronto, ON Canada, Oct. 1993. National Research Council of Canada.