Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components

Victor R. Basili
University of Maryland
Computer Science
College Park, MD
20742 USA
basili@cs.umd.edu

Steven E. Condon
CSC
10110 Aerospace Rd.
Lanham-Seabrook, MD
20706 USA
scondon@csc.com

Khaled El Emam
Fraunhofer IESE
Sauerwiesen 6
D-67661
Kaiserslautern
Germany
elemam@iese.fhg.de

Robert B. Hendrick
CSC
10110 Aerospace Rd.
Lanham-Seabrook
MD
20706 USA
bhendric@cscmail.csc.com

Walcelio Melo
CRIM
1801 McGill College
Montreal, Quebec
Canada
H3A 2N4
wmelo@crim.ca

ABSTRACT

In this paper we characterize and model the cost of rework in a Component Factory (CF) organization. A CF is responsible for developing and packaging reusable software components. Data was collected on corrective maintenance activities for the Generalized Support Software reuse asset library located at the Flight Dynamics Division of NASA's GSFC. We then constructed a predictive model of the cost of rework using the C4.5 system for generating a logical classification model. The predictor variables for the model are measures of internal software product attributes. The model demonstrates good prediction accuracy, and can be used by managers to allocate resources for corrective maintenance activities. Furthermore, we used the model to generate proscriptive coding guidelines to improve programming practices so that the cost of rework can be reduced in the future. The general approach we have used is applicable to other environments.

Keywords

Software Process Improvement, cost of rework, software metrics, classification models, prediction models.

INTRODUCTION

Previous research has shown that software reuse has a great potential to improve software development productivity and product quality [6][25][19]. For example, effective reuse of knowledge, processes and products from previous experience can decrease software development cost, reduce project delivery time and improve software quality [5][13].

However, reuse will not just happen—rather, components must be designed for reuse, and organizational elements must be created to enable projects to take advantage of the reusable software artifacts [2][11][26].

To facilitate the packaging and reuse of software development experience, an infrastructure called the Component Factory (CF) has been proposed [4]. The CF is a separate entity from the organization that produces applications. The CF is responsible for developing and packaging reusable software components. It creates and maintains a software component repository for future reuse and supplies reusable components to the development organization upon demand.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copy-right notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee

ICSE 97 Boston MA USA

Copyright 1997 ACM 0-89791-914-9/97/05 ..$3.50

Several studies have empirically examined the characteristics of reusable components. For example, [22] investigated new versus reused code in a large collection of FORTRAN projects to analyze the pros and cons of creating a component from scratch versus modifying an existing component. Also in [25], eight medium scale Ada projects were assessed with respect to the defects found in newly developed and reused components. However, none of these works were concerned with software components that were developed exclusively for reuse. As far as we know, studies of reuse have focused on the side of the project organization, which reuses the components, rather than on the side of the CF, which creates the components. The primary reason for this different focus appears to be that not many software companies have a CF set up to develop and maintain reusable software components. Another potential explanation is that the few existing CFs have not collected sufficient data allowing them to evaluate the different aspects of the development and maintenance of reusable components.

In this paper we present a study that characterizes and models the cost of rework for a library of reusable components. This library, known as the Generalized Support Software (GSS) reuse asset library, is located at the Flight Dynamics Division (FDD) of NASA's Goddard Space Flight Center (GSFC). Component development began in 1993. Subsequent efforts focused on generating new components to populate the library and on implementing specification changes to satisfy mission requirements. The first application using this library was developed in early 1995.

The asset library currently consists of 921 Ada83 components totaling approximately 515 KSLOC. Based on a review of the first 58 GSS error correction reports, 102 of these 921 components have required error correction one or more times. We first characterize the 58 error correction reports in terms of source of error, class of error (both defined below), effort required to isolate the error, and effort required to correct the error. We then use a machine learning

algorithm, C4.5 [21], to construct a model for approximate prediction of the cost of rework ("high" or "low"), using internal source code metrics of the components that are changed. The prediction model can help managers of the GSS asset library in the decision-making process by providing them with guidelines for predicting where corrective maintenance resources will most be needed. The model also consists of a set of easily interpretable coding guidelines that can be applied in improving current practices in order to reduce the cost of future rework. We expect that the process used to model rework in the GSS environment can be used in other environments to provide equally effective prediction models and coding guidelines that are appropriate for those environments.

In [16], various modeling techniques were used to predict maintenance productivity. In that article, the only product metric that was considered was a software size measure based on LOC. In [8], a machine learning algorithm was also used to predict the cost of rework in an Ada environment using internal product metrics. Unlike the components we have studied, however, the components analyzed in [8] were developed to satisfy specific application requirements. The current paper is, to our knowledge, the first that applies machine learning techniques to help manage the maintenance of reusable components, and to improve the way these components are produced in order to reduce maintenance costs within a CF.

The paper is organized as follows. It first presents the framework in which this study was conducted: the FDD, the Software Engineering Laboratory (SEL), and the GSS domain engineering and application deployment processes. The paper then presents the method for data collection and analysis. Then, the results of our analysis, including descriptive statistics that characterize the components and a predictive model of rework effort, are presented. We conclude the paper with a summary and directions for future work.

ENVIRONMENT OF THE STUDY

The FDD

GSFC manages and controls NASA's Earth-orbiting scientific satellites and also supports human space flight. For fulfilling flight dynamics responsibilities for both of these complex missions, the FDD developed and now maintains over 100 different software systems, ranging in size from 10 thousand source lines of code (KSLOC) to 300 KSLOC, and totaling approximately 4.5 million SLOC. This software covers three separate subdomains of the FDD mission: mission planning, orbit determination, and attitude[1] determination.

To increase the amount and type of reuse, and at the same time to drastically reduce the cycle time needed to develop and test new software systems, the FDD embarked on the GSS Domain Engineering Process in 1993. This process achieves rapid deployment by utilizing an object-oriented architecture in which the reusable assets are the generalized specifications for the reusable software components, as well as the reusable software components themselves (written in Ada83). Adopting this architecture and process results in a paradigm shift from *developing* software applications to *configuring* software applications. The GSS reuse asset library is the software component repository examined in this paper.

The SEL

The Software Engineering Laboratory began in 1976 with the goals of understanding the software process and product in the FDD, determining the impact of available technologies, and infusing the identified/refined methods, techniques, and products back into the environment. The approach has been to identify technologies with potential, apply them, and study their effect, based on studying the impact of the changes on such issues as cost, reliability, and quality. The participating organizations are the FDD, the University of Maryland, and Computer Sciences Corporation.

Over the years, the SEL has investigated numerous techniques and methods in over a hundred projects to understand and improve the software development process and product in their environment [20]. The result of this legacy is an organization and personnel that are quite interested in experimentation with new technologies and not averse to change. They are also a part of an environment that is quite successful at the type of work in which they are involved.

The approaches used for learning include the concept of the Experience Factory (EF). The focus of the EF in the SEL is on collecting metrics and lessons learned from standard projects and from special experiments, and then analyzing these data and packaging them into guide books, models, and training courses that can be spread to all areas of the development organization. The EF is different from the Project Organization (PO) which focuses on the development and maintenance of applications. Their relationship is depicted in Figure 1. The SEL EF has developed and packaged:

- resource models and baselines (e.g., local cost models, resource allocation models)
- change and defect baselines and models (e.g., defect prediction models, types of defects expected for the application)
- project models and baselines (e.g., actual vs. expected product size)
- process definitions and models (e.g., process models for Cleanroom, Ada waterfall model)
- method and technique evaluations (e.g., best method for finding interface faults)
- products and product parts (e.g., Ada generics for simulation of satellite orbits)
- quality models (e.g., reliability models, defect slippage models, ease of change models), and

1. The term "attitude" refers to a spacecraft's orientation in space.

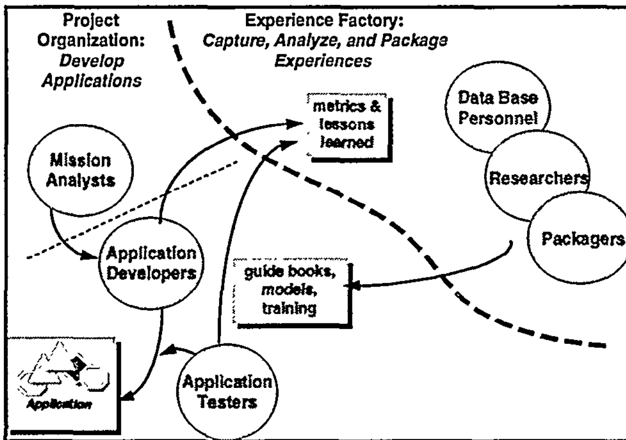

Figure 1: The relationship between the Experience Factory and the Project Organization.

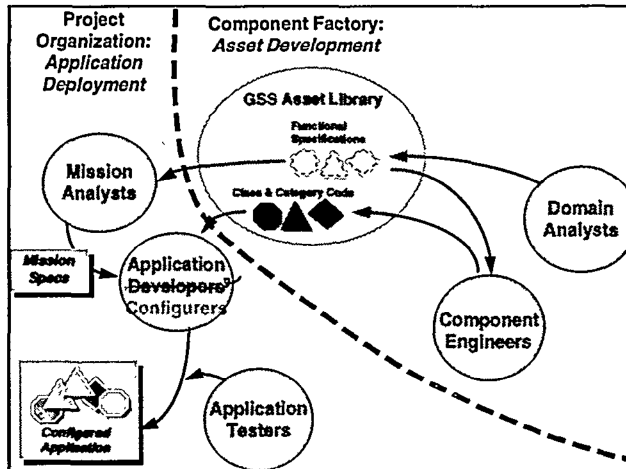

Figure 2: The relationship between the Component Factory and the Project Organization.

- lessons learned (e.g., risks associated with an Ada development).

These models are built to understand the local environment, identify areas for improvement, attempt improvement via change, and form bases for evaluating that change against goals.

The Component Factory (CF) organization is a sub-organizational structure of the EF—an addition to the traditional EF. The CF focuses on generating a configuration architecture and reusable components, based on learning over time. This learning is in the form of analysis and synthesis of what is most effective for reuse (as well as what is expected to be needed for configuring applications) for the future development of products in a certain class. To staff a CF, some members of the PO functionally become members of the CF, although they may continue to think of themselves still as PO members. (See Figure 2.) That is, some mission analysts and application developers become domain analysts for the CF, and some application developers become component engineers for the CF. The domain analysts design the architecture and class specifications of the reuse asset library. The component engineers then construct the reusable class components. The PO takes advantage of this architecture and asset library to configure new systems. The PO's mission analysts now compare mission requirements to the asset library's functional specifications and produce a *mission specification* document that tells the PO's application configurers—application developers are no longer needed—how to configure the desired system from the reuse library assets. The traditional elements of the EF, together with the CF staff, then study how effective this process and the asset library have been for future improvement

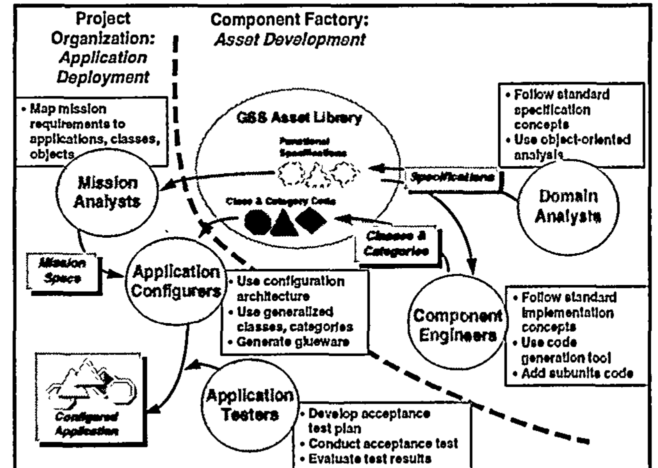

Figure 3: The GSS domain engineering and application deployment process.

The GSS Process

The activites of the CF and the PO in the GSS domain engineering and application deployment process are shown in more detail in Figure 3. The process relies on five functionally distinct teams, although some personnel may overlap between teams (particularly between the component engineers and the application configurers). The domain analysts write the class and category specifications. The component engineers code the classes and categories that, together with the specifications, make up the GSS reuse asset library. The mission analysts analyze the mission requirements and specify which classes need to be used for a given mission and how they should be configured. The application configurers configure the desired mission applications from the available classes and categories in the GSS reuse asset library, instantiate the generics, and perform integration testing of the application. The application testers conduct acceptance testing of the configured mission application.

DATA COLLECTION AND ANALYSIS METHOD

Definitions

Errors are defects in the human thought process that are made while trying to understand and communicate given information, solve problems, or use methods and tools. Faults are concrete manifestations of errors within the

software.

In this study, an *error* is represented by a single software Change Request Form (CRF) [15] filled by developers and configurers to institute and document a change to one or more components. A CRF results in modifications to one or more components in the reuse asset library. CRFs are also generated for enhancements, requirements changes, and adaptation. The current paper examines only error correction CRFs.

A *fault* pertains to a single component and is evidenced by the physical change of that component in response to a particular error CRF. In this study, we define a *component* as as an Ada file in configuration management. A *faulty* component version becomes a *fixed* component version after it is corrected. We are only interested in the faulty component versions.

Data Collection

We collected data on: (1) error identification and error correction (which follow initiation of a CRF), including the names and version numbers of the source code components that had faults in them, and (2) source code metrics characterizing these particular components.

Between 9th March 1994 and 21st September 1995, a total of 58 GSS error correction CRFs were generated, meaning 58 errors were identified. (In addition, 96 additional GSS CRFs were generated for requested enhancements, adaptations, and requirements changes.) Most of the GSS error correction CRFs were initiated by configurers, who uncovered problems during instantiation of the Ada generics and integration testing of the configured application, prior to turning over the configured application to acceptance testing. A very small minority of the CRFs—perhaps ten percent—were initiated by a maintainer of the reuse asset library following the report of a failed application test item by the independent tester group during conductance of acceptance testing of the application..

The CRF data analyzed by our study consisted of (1) the classification of errors by source and class, (2) the names of components changed to correct the errors, (3) the effort expended to isolate all faults associated with the error, and (4) the effort required to correct all of these faults. Each of these is described below.

Isolation and correction effort was measured on a 4-point ordinal scale: 1 hour, from 1 hour to 1 day, from 1 to 3 days, and more than 3 days. In addition, the maintainer provides the source of the error (requirements, functional specification, design, code, or previous change). Once an error is found during configuration and testing, the maintainer finds the cause of the error, locates where the modifications are to be made, and determines that all effects of the change are accounted for. Then the maintainer modifies the design (if necessary), code, and documentation to correct the error. Once the maintainer fixes the error, the maintainer provides the names of the components changed (in our case the faulty components). The maintainer also specifies the class of the error (initialization, internal/ external interface, user interface, database, algorithm, etc.).

The Amadeus tool [1] was used to extract source code metrics from all faulty component versions. A description of the source code metrics that were found useful is given in the results section of this paper. If after the extraction of some metrics it was found that they had zero variation (e.g., the number of Goto's), we excluded these metrics from further analysis.

Data Analysis: Characterization

The first data analysis task was to characterize or describe the errors. The objective of this characterization is to understand better the nature of the errors and how they are distributed. For this, basic pie charts were used. Furthermore, basic bivariate analysis using contingency tables and chi-square tests [24] was conducted to identify if there were any relationships between the source and class of errors and the rework effort.

Since the contingency tables tended to be sparse in some instances (i.e., cell frequencies approaching zero), we dichotomized each of the isolation and correction cost variables. We therefore considered isolation or correction effort of 1 hour as *Low*, and effort greater than 1 hour to be *High*.

Data Analysis: Modeling

A cost of rework model should allow: (1) the prediction of which components are likely to be associated with costly rework, and (2) provide programming guidelines that can be used to prevent costly rework in the future. The cost of rework is measured as the *total* effort taken to isolate and correct an error.

Unit of Analysis

The unit of analysis for developing the model is a faulty component version. During rework, a total of 118 changes were made to 102 components to fix these 58 errors. Four of the components were changed three times (i.e., on three different CRFs), 8 components were changed twice, and the remaining 90 were changed only once.

Approximately 75% of the components in the library are generated using a code generator. When software changes are necessary, maintainers do not make changes directly to the outputs of the code generator. Instead, the inputs to the code generator are changed, and new versions of the output components are generated. Given that rework effort is only directly affected by the characteristics of the component versions that are actually changed by the maintainers, component versions that are automatically regenerated by the code generator should not be included in our analysis. Where the components associated with a CRF include the input to the code generator as well as the output component, we excluded the modified output versions in our analysis. This leaves a total of 76 faulty component versions which are the basis of our analysis.

Model Specification

The model that we developed identifies component versions that are associated with costly rework rather than trying to predict the exact effort for reworking a component version. We therefore use the characteristics of a faulty component version as input into the model, and the total rework effort

for the error as the output of the model. Given that the model we developed is a classification model, it classifies a component version into ones of two rework cost categories: *Low Cost* and *High Cost*. (Note that these categories are different from the one described in the "Characterization" paragraph above because, for the model we are interested in *total* rework effort, while in characterization we look at isolation and correction separately.) This allows the model to predict whether a component version is associated with a costly, or otherwise, error.

Modeling Technique

The modeling technique that we used is a machine learning algorithm called C4.5 [21]. The C4.5 algorithm partitions continuous attributes, in our case the internal product metrics, finding the best threshold among the set of training cases to classify them on the dependent variable. As well as being useful for prediction, the generated tree provides decision rules characterizing component versions that fall into each one of the two rework cost categories.

We chose this technique because the models are straightforward to build and are also easy to interpret. In addition, this class of modeling techniques has been used in the software engineering literature to build prediction models [23], and therefore there already is some familiarity with it. Of course, other classification techniques, e.g., Optimized Set Reduction [9] or logistic regression [6], could have been used. However, our goal here is not to compare classification techniques.

Potential Application of the Model

A prediction identifying component versions that are going to be associated with costly errors can help managers allocate resources for the maintenance activities. The availability of rules as part of the model can help prevent high rework cost in the maintenance environment. For example, rules that characterize high rework cost can be treated as *proscriptive* programming guidelines for developing future components. It is on proscriptive rules that we focus in this study.

It should be noted, however, that the model does *not* identify which component versions in the asset library are likely to have faults, only which of the faulty versions should be more or less expensive to isolate and correct. Application of such predictions assumes that the manager knows *beforehand* which components are likely to contain a fault. Models for the prediction of fault-prone Ada components in the SEL environment have been developed in the past [9]. Once a component version has been identified as potentially fault-prone, then it is possible to predict the cost of rework category when fixing an error that leads to faults in that version. Using this additional information, a manager can improve the resource allocation for maintenance.

Dependent Variable

To build a classification model, we dichotomize our dependent variable, which is the total cost of rework. We converted the four effort categories into average values following [3]. We assumed an 8 hour day, and took the average value for each of the categories of rework. Therefore, the category of "1 Hour" was changed to 0.5 hours, the category of "1 hour to 1 Day" was changed to 4.5 hours, the category of "from 1 to 3 Days" was changed to 16 hours, and the category of "more than 3 Days" was changed to 32 hours. We then summed up these values for isolation and correction costs. This gives us an average overall rework cost. The median of total rework cost per CRF was 5 hours, and we used that as the cutoff point for dichotomization. Based on this dichomotomization, we have 33 component versions that were associated with errors requiring a low cost of rework and 43 that required a high cost of rework.

Independent Variables

Internal product metrics have been widely used to predict quality attributes such as productivity and software quality [14]. Here, we are interested in studying the use of internal product metrics of the faulty GSS component versions to predict the cost of rework. Previous research investigated the use of the characteristics of the change as the basis for the prediction of correction effort [10], however, the characteristics of the change are usually not available before the change is actually made (or at least not before isolation of the error). We only wanted to use information that would be available before isolation in order to develop a model for predicting total rework effort.

Evaluation of the Model

To evaluate the model, we need criteria for evaluating the overall model accuracy and for evaluating the strength of the rules. Evaluating model accuracy tells us how good the model is expected to be as a predictor. Evaluating the strength of the rules tells us the extent to which we can trust these rules as programming guidelines.

Evaluating Prediction Accuracy

Three criteria for evaluating the accuracy of predictions are the predictive validity criterion, and measures of correctness and completeness. These are defined below with reference to Table 1. Table 1 shows symbols for frequencies.

A criterion of prediction validity has been presented in [17]. This basically involves laying out the frequencies as in Table 1, and calculating the chi-square statistic. If the value is larger than a critical value then it is claimed that the model has predictive validity. The authors state that a model that does not meet the criterion of predictive validity should be rejected. This does not necessarily mean that a model that meets the predictive validity criteria should be accepted (it would be easy to demonstrate that if the classification model predicted all High Cost components as Low Cost and vice versa - i.e., very high misclassification - it would still have high predictive validity). We use this criterion to determine whether there is any association between the real rework cost of a component and its actual rework cost.

		Predicted Rework Cost	
		Low Cost	High Cost
Real Rework Cost	Low Cost	n_{11}	n_{12}
	High Cost	n_{21}	n_{22}

Table 1: Evaluating the accuracy of predicted classifications.

Correctness is defined as the percentage of component versions that were predicted to be costly to rework and were actually costly to rework. We want to maximize correctness because if correctness is low, then the model is identifying more component versions as being costly to rework when they really are not costly to rework, which could lead to an over-allocation of resources to making changes (i.e., wastage).

$$\text{Correctness} = \left(\frac{n_{22}}{n_{12} + n_{22}} \right) \times 100$$

Completeness is defined as the percentage of those component versions costly to rework and were predicted to be costly to rework. We want to maximize completeness because as completeness decreases, more versions that were costly to rework are mis-identified as not costly to rework, which would lead to a shortage of resources for making changes..

$$\text{Completeness} = \left(\frac{n_{22}}{n_{21} + n_{22}} \right) \times 100$$

In order to calculate values for correctness and completeness, we used a V-fold cross-validation procedure [7]. For each observation X in the sample, a model was developed based on the remaining observations (sample - X). This model was then used to predict whether observation X will have high rework or low rework. This validation procedure is commonly used when data sets are small.

Evaluation of Rules

The generated model from all 76 versions is also useful for providing proscriptive guidelines to programmers. The guidelines inform the programmers of the characteristics of faulty components that tend to require costly rework. By producing components that do not have these characteristics, there is a greater chance that components will be produced that are not costly to rework. There are two ways for evaluating such rules. First by measuring the number of cases that a rule classified correctly. Second, by appeal to the intuition of programmers in the environment (i.e., do the rules make sense to them).

RESULTS

Characterizing Errors

Distribution of Errors by Error Source

Figure 4 shows the overall distribution of errors (the 58 errors) by error source. Requirements and functional specification errors are those triggered by a misunderstanding of user requirements, and are introduced into the system by the process of transforming user requirements into project requirement specifications. Design errors are those introduced in the process of transforming requirements and specifications into detailed (component-level) design. Coding errors are those that occur when transforming the detailed design to code, such as mistyping a variable name, incorrectly coding an assignment statement, or incorrectly coding the exit criteria of a loop. Finally, errors resulting from a previous change are those that were not in the system until some other change was implemented (in which case the implementation of the previous change did not consider all of its possible effects, or the change was simply implemented incorrectly).

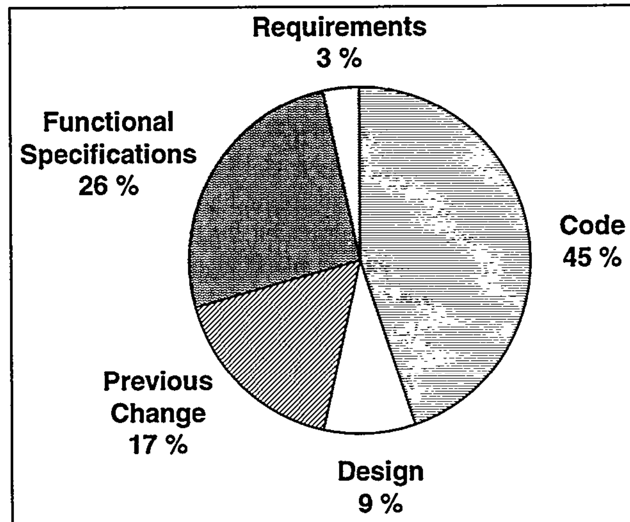

Figure 4: Distribution of errors by source.

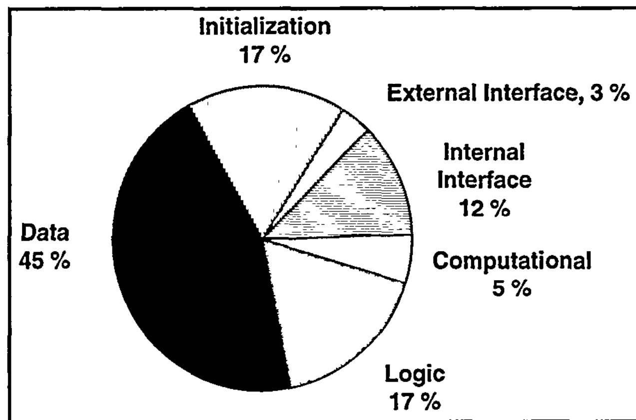

Figure 5: Distribution of errors by class.

Coding errors are responsible for approximately half of the errors found during acceptance testing (45%), followed by errors from requirements and functional specifications (29%), previous changes (17%), and finally design (9%).

It is interesting to note the small amount of design errors compared with requirements, specification, and coding errors. In part, this stems from the fact that most of the "design" of the GSS library is done during the specification phase. The object classes and the relationship between such classes of the three types of applications developed in the FDD (orbit, attitude and mission support) are, in fact, defined during the requirements analysis phase. The description of the methods of GSS classes are also done during the analysis.

Distribution of Errors by Error Class

The components in the library are based on generalizations of existing algorithms that were previously used in earlier systems. Therefore logic and computational errors are expected to be low (17% and 5% respectively as seen in Figure 5).

Initialization errors are responsible for 17% of the errors found during acceptance testing. (Initialization errors are those which result from an incorrectly initialized variable, failure to reinitialize a variable, or because a necessary initialization was missing; failure to initialize or reinitialize a data structure properly upon a component's entry/exit is also considered an initialization error). Once an application is created using the component library, a minimal set of integration tests are run. Particularly for an initial version of an application, this can result in a large number of initialization errors since this would be the first time the components have been configured in this fashion.

Data (value or structure) are responsible for the largest proportion of errors caught by the configurers and testers (see Figure 5). Data errors are those provoked by any error resulting from an incorrect use of a data structure. Examples of data errors are the use of incorrect subscripts for an array, the use of the wrong variable, the use of the wrong unit of measurement, or the inclusion of an incorrect declaration of a variable local to the component. One potential explanation for the large incidence of Data errors is that the Ada compiler catches a large proportion of the errors that would fall in the other categories, but many common Data errors will pass through compilation. This could be, for example, specifying a variable as POSITIVE instead of NATURAL.

Characterizing the Cost of Rework

Distribution of Errors By Cost of Isolation and Correction

Most of the GSS errors had a low isolation cost (60%) and a low correction cost (64%). It can be hypothesized that the design of the GSS architecture and the use of coding standards help reduce the time necessary to isolate errors, as well as the application of object-oriented design principles. Another explanation for the relatively low rework costs in general is that the people responsible for correcting errors in the GSS components have participated in the development of these components. They have, therefore, a good understanding of the design and realization strategies implemented into the code.

It should be noted that the median number of components changed for each CRF is 1 (maximum is 6), and the median number of other components examined is zero (with a maximum of 5). To test the hypothesis that the number of changed and examined components is related to the cost of isolation and correction, we used the Mann-Whitney U test [24]. No difference was found for the number of components examined when isolation cost was considered. When considering correction cost, it was found that more components are changed for high correction cost CRFs compared to low cost CRFs (at an alpha level of 0.05). No difference was found for number of components examined and correction cost.

Impact of Error Source on Rework Effort

Table 2 shows the distribution between the categories of error isolation cost and the error source. The contingency table contains the frequency of CRFs in each cell and the percentage of the total. We combined the Requirements and Functional Specification sources together into one "Analysis" category to avoid having expected frequencies less than one in the table. Likewise, Table 3 shows the distribution between the categories of error correction cost and the source of error.

Observation of the table indicates that for analysis sources, the isolation and correction costs tend to be low. We used the Pearson chi-square statistic to determine if there is a general association between source and rework cost. The probability values for both the isolation cost and the correction cost table were not significant at the 0.05 alpha level.[1]. Therefore,

1. The approximation of the X^2 statistic to the chi-square distribution assumes that expected frequencies are not too small. This is usually interpreted to mean having at least 20% of expected frequencies greater than 5 and no cell having an expected frequency less than 1 for tables with degrees of freedom greater than 1 [12]. However, it has been suggested that the conventional chi-square statistic may be used for 2xc tables where all expected frequencies are as low as 1 [18].

there is no association between source of error and isolation nor correction cost.

	Code	Design	Analysis	Previous Change	Total
HIGH Isolation Cost	13 22.4%	2 3.45%	4 6.9%	4 6.9%	23
LOW Isolation Cost	13 22.4%	3 5.17%	13 22.41%	6 10.34%	35
Total	26	5	17	10	58

Table 2: Relationship between error source and isolation cost.

	Code	Design	Analysis	Previous Change	Total
HIGH Correction Cost	9 15.52%	3 5.17%	5 8.62%	4 6.9%	21
LOW Correction Cost	17 29.31%	2 3.45%	12 20.69%	6 10.34%	37
Total	26	5	17	10	58

Table 3: Relationship between error source and correction cost.

	Computational	Data	Initialization	Interface	Logic	Total
HIGH Isolation Cost	1 1.72%	11 19%	3 5.17%	2 3.45%	6 10.34%	23
LOW Isolation Cost	2 3.45%	15 25.86%	7 12%	7 12%	4 6.9%	35
Total	3	26	10	9	10	58

Table 4: Relationship between error class and isolation cost.

	Computational	Data	Initialization	Interface	Logic	Total
HIGH Correction Cost	1 1.72%	11 18.97%	2 3.45%	4 6.9%	3 5.17%	21
LOW Correction Cost	2 3.45%	15 25.86%	8 13.79%	5 8.62%	7 12%	37
Total	3	26	10	9	10	58

Table 5: Relationship between error class and correction cost.

		Predicted Rework Cost: Low Cost	Predicted Rework Cost: High Cost	
Real Rework Cost	Low Cost	23	10	33
	High Cost	12	31	43
		35	41	76

Table 6: Predicted versus real rework categories.

Impact of Error Class on the Cost of Rework
Table 4 shows the distribution between the class of error and the isolation cost. We combined the Internal and External Interface categories to avoid having cells with expected frequencies less than one. The relationship between source and correction cost is depicted in Table 5. It can be observed from the tables that interface errors tend to cost less to isolate, and initialization errors tend to cost less to correct. Chi-square tests however do not identify any statistically significant association for either of the two tables.

Modeling the Cost of Rework
Table 6 shows the relationship between real and predicted rework. The predictive validity criterion for the contingency table presented in Table 6 is met at a one-tailed alpha level of 0.05. The values of correctness and completeness are shown in Figure 6. We found that correctness was 76% and completeness 72%. These values were perceived to be sufficient for decision making, especially when combined with expert judgment.

In this paper we are concerned with rules that characterize component versions that are costly to rework. The proportion of components that match the rule and are classified correctly by the rule give us a measure of how accurate a particular rule is. The model we developed had three interpretable rules for classifying high rework cost component versions. These are shown in Figure 7. For engineers involved with the GSS asset library, the rules were perceived to be intuitive in the sense that they express the fact that "more complicated things are more likely to cost more to correct." Moreover, the rules formalize the characteristics of the more complicated component versions.

The three rules can be used as maximal thresholds when developing new components. In some cases, there may be good design reasons for a component to exceed the threshold(s). Therefore the rules ought not be interpreted as strictly proscriptive. If a new component matches one or more of the rules, then the developer can decide whether it needs to be changed to reduce its potential for being associated with an error that is costly to isolate and correct.

Figure 8 shows the 3 internal product metrics that were found useful in developing this model. These 3 metrics were automatically selected by C4.5 from the set of metrics provided by Amadeus.

Correctness	76% (31/41)
Completeness	72% (31/43)

Figure 6: Correctness and completeness results for the prediction model.

Rule(s)	Accuracy
FunctionCalls > 38	100%
DeclarationStatements > 59	90%
ProgrammerExceptionsUsed > 2	83%

Figure 7: Proscriptive coding rules and their accuracy.

Metric Name	Brief Description
FunctionCalls	The number of function calls.
DeclarationStatements	The number of declaration statements, including those with and without initialization.
ProgrammerExceptionsUsed	The number of exceptions used in the file.

Figure 8: Description of the metrics that were found useful for building the model.

The proscriptive guidelines provided in Figure 7 were found from error data for a specific reusable components library. Caution should be exercised in attempting to generalize these rules beyond this context and applying them in a different environment. The overall approach we have used, however, can easily be generalized to other contexts. For example, after collecting the appropriate data, another organization could develop models for prediction and for producing coding guidelines to manage and reduce rework effort.

CONCLUSIONS

In this paper we reported on a study to model and understand the cost of rework in a library of reusable software components. We described how rework costs are distributed during the error correction process, and developed a model to predict the component versions that are associated with errors that are costly to rework. The model was also used to develop proscriptive coding rules that can be used by programmers as guidelines to reduce the cost of rework in the future.

Extensions of this work would include developing models for predicting components that have a high risk of faults (to help managers focus testing and inspections) and that can also be used to provide guidelines to programmers. We have used a specific set of internal product metrics for developing the model. These metrics tended to be counts of elements of a component. A different set of metrics that better characterize the structure and design of components may improve the predictive quality of the model, and also would provide guidelines for improving design practices.

Furthermore, it would be informative to compare models where cost of rework is the dependent variable with models where risk of fault is the dependent variable to determine if the derived guidelines from the two models are complementary or contradictory.

ACKNOWLEDGEMENTS

The authors wish to thank Lionel Briand, Steve Burke, Minna Makarainen, Mark Nicholson, Mari Sykes, and Carolyn Seaman for their help in data collection and preparation, comments on this work and reviews of earlier versions of this paper. The work reported in this paper was partially funded through NASA grant 01-5-26393 and NASA contract NAS5-31000 (subcontractor no. HQ-001057). Walcelio Melo was sponsored by Bell Canada, Software Quality Group, for the work reported in this paper.

REFERENCES

1. Amadeus Software Research Inc.: "Getting Started with Amadeus." Amadeus Measurement System. 1994.
2. R. Banker, R. Kauffman and D. Zweig: "Repository Evaluation of Software Reuse." In *IEEE Transactions on Software Engineering*, 19(4):379-389, April 1993.
3. V. Basili and B. Perricone: "Software Errors and Complexity: An Empirical Investigation." In *Communications of the ACM*, 27(1):42-52, January 1984.
4. V. Basili, and H.D. Rombach: "The TAME Project: Toward an Improvement-oriented Software Environment." In *IEEE Transactions on Software Engineering*, 14(6):758-773, June 1988.
5. V. Basili, and H.D. Rombach: "Support for Comprehensive Reuse." In *Software Engineering Journal*, 6(5):303-316, September 1991.
6. V. Basili, L. Briand and W. Melo: "A Validation of Object-Oriented Design Metrics as Quality Indicators." In *IEEE Transactions on Software Engineering*, December 1996.
7. L. Breiman, J. Friedman, R. Olshen and C. Stone: *Classification and Regression Trees*. Published by Wadsworth, 1984.
8. L. Briand, W. Thomas, and C. Hetmanski: "Modeling and Managing Risk Early in Software Development". In *Proceedings of the International Conference on Software Engineering*, pages 55-65, 1993.
9. L. Briand, V. Basili and C. Hetmanski: "Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components." In *IEEE Transactions on Software Engineering*, SE-19 (11):1028-1044, 1993.
10. L. Briand and V. Basili: "A Classification Procedure for the Effective Management of Changes during the Maintenance Process". In *Proceedings of the International Conference on Software Maintenance*, pages 328-336, 1992.
11. G. Caldiera, and V. Basili: "Identifying and qualifying reusable software components." In *IEEE Software*, pp.61-70, February 1991, .
12. W. Cochran: "Some Methods for Strengthening the Common X^2 Tests". In *Biometrics*, 10:417-451, 1954.
13. W. Frakes, and S. Isoda: "Success factors of systematic reuse." In *IEEE Software*, pp.15-19, September 1994.
14. W. Harrison: "Software measurement: a decision-process approach." *Advances in Computers*, 39:51–105. 1994.
15. G. Heller, J. Valett and M. Wild: *Data Collection Procedure for the Software Engineering Laboratory (SEL) Database*. Technical Report SEL-92-002, Software Engineering Laboratory, 1992.
16. M. Jorgensen: "Experience with the Accuracy of Software Maintenance Task Effort Prediction Models". In *IEEE Transactions on Software Engineering*, 21(8):674-681, August 1995.
17. F. Lanubile and G. Visaggio: "Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned". Technical Report ISERN-96-03, International Software Engineering Research Network, 1996.
18. R. Lewontin and J. Felsenstein: "The Robustness of Homogenity Tests in 2xN Tables". In *Biometrics*, 21:19-33, 1965.
19. W. Lim: "Effects of Reuse on Quality, Productivity, and Economics." In *IEEE Software*, pp.23-30, September 1994.
20. F. McGarry, R. Pajerski, G. Page, S. Waligora, V. Basili, and M. Zelkowitz: *An Overview of the Software Engineering Laboratory*. Technical Report SEL-94-005, Software Engineering Laboratory, 1994.
21. J. R. Quinlan: *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Sao Mateo, CA, 1993.
22. R. Selby: "Empirically Analyzing Software Reuse in a Production Environment". In W. Traca (ed.) *Software Reuse: Emerging Technology*. IEEE Press, 1988,
23. R. Selby and A. Porter: "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis." In *IEEE Transactions on Software Engineering*, 14(2):1743-1747. 1988.
24. S. Siegel and N. Castellan: *Nonparametric Statistics for the Behavioral Sciences*. McGraw Hill, 1988.
25. W. Thomas, A. Delis, and V. Basili: "An Analysis of Errors in a Reuse-oriented Development Environment." Technical Report, University of Maryland, 1995.
26. C. Wohlin, and P. Runeson: "Certification of Software Components." In *IEEE Transactions on Software Engineering*, 20(6):494-499, June 1994.