Identifying Reasons for Software Changes Using Historic Databases

Audris Mockus
Bell Laboratories
Software Production Research Department
263 Shuman Blv. Rm. 2F-319
Naperville, IL 60566
audris@research.bell-labs.com

Lawrence G. Votta
Senior Member of Technical Staff
High Availability Platform Development
Motorola, Inc.
Arlington Heights, IL 60004 USA
lvotta1@email.mot.com

Abstract

Large scale software products must constantly change in order to adapt to a changing environment. Studies of historic data from legacy software systems have identified three specific causes of this change: adding new features; correcting faults; and restructuring code to accommodate future changes.

Our hypothesis is that a textual description field of a change is essential to understanding why that change was performed. Also, we expect that difficulty, size, and interval would vary strongly across different types of changes.

To test these hypotheses we have designed a program which automatically classifies maintenance activity based on a textual description of changes. Developer surveys showed that the automatic classification was in agreement with developer opinions. Tests of the classifier on a different product found that size and interval for different types of changes did not vary across two products.

We have found strong relationships between the type and size of a change and the time required to carry it out. We also discovered a relatively large amount of perfective changes in the system we examined.

From this study we have arrived at several suggestions on how to make version control data useful in diagnosing the state of a software project, without significantly increasing the overhead for the developer using the change management system.

1 Introduction

The traditional approaches to understanding the software development process define specific questions, experiments to answer those questions, and instrumentation needed to collect data (see, e.g., the GQM model [2]). While such an approach has advantages (i.e., in some cases defines a controlled experiment), we believe that a less intrusive and more widely applicable approach would be to obtain the fundamental characteristics of a process from the extensive data available in every software development project. To ensure that our methods could be easily applied to any such project, we used data from a version control system. Besides being widely available, the version control system provides a data source that is consistent over the duration of the project (unlike many other parts of the software development process). Our model of a minimal version control system (VCS) associates date, time, size, developer, and textual description with each change.

Implicit in our approach is the assumption that we consider only software process properties observable or derivable from the common source — VCS. Because VCSs are not designed to answer questions about process properties (they are designed to support versions and group development), there is a risk that they may contain minimal amounts of useful process information despite their large size. There may be important process properties that can not be observed or derived from VCSs and require more specialized data sources.

The quantitative side of our approach focuses on finding main factors that contribute to the variability of observable quantities: size, interval, quality, and effort. Since those quantities are interdependent, we also derive relationships among them. We use developer surveys and apply our methods on different products to validate the findings.

This work exemplifies the approach by testing the hypothesis that a textual description field of a change is essential to understand why that change was performed. Also, we hypothesize that difficulty, size, and interval would vary across different types of changes.

To test our hypotheses, we analyzed a version control database of a large telecommunications software system (System A). We designed an algorithm to classify automatically changes according to maintenance activities based on the textual description field. We identified three primary

1063-6773/00 $10.00 © 2000 IEEE

reasons for change: adding new features (adaptive), fixing faults (corrective), and restructuring the code to accommodate future changes (perfective), consistent with previous studies, such as, [22]. It should be noted that our characterizations of adaptive and perfective maintenance are not entirely consistent with the literature (see, e.g., [9]), where new features requested by a user are considered to be perfective maintenance, while new features required by new hardware or new software interfaces are considered to be adaptive maintenance.

We discovered a high level of perfective activity in the system, which might indicate why it has been so long on the market and remains the most reliable among comparable products. We also discovered that a number of changes could not be classified into one of the primary types. In particular, changes to implement the recommendations of code inspections were numerous and had both perfective and corrective aspects. The three primary types of changes (adaptive, corrective, perfective), as well as inspection rework, are easily identifiable from textual description and have strikingly different size and interval. The types of changes and frequency of changes have been found to be related to the age and size of a software module in [14].

To verify the classification we did a survey where we asked developers of System A to classify their own recent changes. The automatic classification was in line with developer opinions. We describe methods and results used to obtain relationships between the type of change and its size or interval.

We then applied the classifier to a different product (System B) and found that the change size and interval varies much less between products than between types of changes. This indicates that size and interval might be used to identify the reason for a change. It also indicates that this classification method is applicable to other software products (i.e., it has external validity). We conclude by suggesting new ways to improve the data collection in the configuration management systems.

Sections 2 describes the System A software product. Section 3 introduces the automatic classification algorithm and Section 4.1 describes the developer validation study. We illustrate some of the uses of the classification by investigating size, interval, and difficulty for different types of changes in Section 5. In Subsection 5.2 the classifier is applied to System B. Finally, we conclude with recommendations for new features of change control systems to allow analysis of the changes and hence the evolution of a software product.

2 Software product data

Our database was version control and maintenance records from a multi-million line real-time software system that was developed over more than a decade. The source code is organized into subsystems with each subsystem further subdivided into a set of modules. Each module contains a number of source code files. The change history of the files is maintained using the Extended Change Management System (ECMS) [16], for initiating and tracking changes, and the Source Code Control System (SCCS) [19], for managing different versions of the files. Our data contained the complete change history, including every modification made during the project, as well as many related statistics.

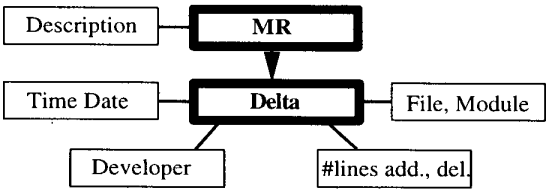

Figure 1. Changes to the code (bold boxes) and associated attributes. Each modification request contains a number of deltas.

Each logically distinct change request is recorded as a Modification Request (MR) by the ECMS (see Figure 1). Each MR is owned by a developer, who makes changes to the necessary files to implement the MR. The lines in each file that were added, deleted and changed are recorded as one or more "deltas" in SCCS. While it is possible to implement all MR changes restricted to one file by a single delta, but in practice developers often perform multiple delta on a singe file, especially for larger changes. For each delta, the time of change, the login of the developer who made the change, the number of lines added and deleted, the associated MR, and several other pieces of information are all recorded in the ECMS database. This delta information is then aggregated for each MR. Each MR has associated English text describing reasons for the change and the change itself. There is no protocol on how and what information is entered, but the text is sufficient for other developers to understand what changes were made and why. A detailed description of how to construct change measures is provided in [17].

In the analysis that follows we use the following measures of size: the number of deltas, numbers of lines of code added, deleted, and unmodified by the change. To obtain these measures we simply count all deltas in a change and add the last three measures over all deltas in a change (each SCCS delta records numbers of lines of code added, deleted, and unmodified). We measure interval of a change by the time lag between the first and the last delta in a change.

We selected a subsystem (System A) for our analysis.

The subsystem contains approximately $2M$ source lines, 3000 files, and 100 modules. Over the last decade it had 33171 MRs, each having an average of 4 deltas. Although it is a part of a larger system, the subsystem functionality is sold as a separate product to customers.

3 Classification of Maintenance Activities

Although the three primary maintenance activities (adaptive, corrective, and perfective) are well known, it is not clear what portion of the total number of changes resulted from each type of activity, or whether additional types of maintenance activities exist. Since the version control data generally does not keep an attribute that would identify the purpose of a change, we looked for this information by analyzing the textual abstract of the change. Information retrieval literature (see [20, 18, 13]) deals with text categorization problems focusing on information retrieval, i.e., obtaining a set of documents relevant to user query, large text documents (change abstracts have only 4 to 20 words), and extensive training collections. Since we are not interested in retrieving abstracts that match a particular user query, but instead in discovering purposes, amounts, and locations (in time and within the source code) of different maintenance activities, we had to design an algorithm that is fine-tuned for this kind of classification of change abstracts.

The classification proceeds in five steps:

1. cleanup and normalization;
2. word frequency analysis;
3. keyword clustering and classification and simple classification rules;
4. MR abstract classification;
5. repetition of the analysis starting from step two on unclassified MR abstracts.

3.1 Normalization Step

In the normalization step, the abstracts were cleaned by removing non-alphanumeric symbols, converting all words to lower case, and obtaining the stem of each word using publicly available WordNet [3] software. This step was done to reduce the number of keywords; for example, *fix*, *fixing*, and *fixes* are all mapped to a single term *fix*. We did not use the synonym capability of WordNet since the relationship between concepts used in software maintenance might differ from the relationships found in the natural language.

3.2 Word Frequency Analysis

The next step consists of word frequency and semantic analysis. We obtained frequencies of all the words in the textual description of maintenance requests and manually classified the most frequent terms as being neutral (e.g., the, for, to, code, etc.), or reflecting a particular type of maintenance activity. We envisioned three primary types of maintenance: fault fixes for keywords such as *fix, problem, incorrect, correct;* new code development for keywords *add, new, modify, update;* and code improvement for keywords *cleanup, unneeded, remove, rework.*

3.3 Keyword Clustering

Since some of the words (especially frequent words) might be used in abstracts describing types of changes not associated with the primary meaning of the keyword, we ensured that keywords have enough discriminating power by using the following procedure:

1. For each selected term we read the description of 20 randomly selected changes that contained the term to see if the abstract describes the change of the same type as that assigned to the keyword.
2. If a term matched less than three fourths of the randomly selected abstracts, the term was deemed to be neutral.

The described decision rule is designed to reject the null hypothesis at 0.05 level that half or less of the MR abstracts containing the term belong to the type assigned to the keyword.

As a result of this activity, we discovered that the term *rework* is frequently used in conjunction with code inspection. The development process in this environment requires formal code inspections for any changes in excess of 50 lines of source code. Code inspection is performed by a team of experts who review the code and recommend changes [7, 8]. Typically, those changes are then implemented by a developer in a separate MR. The purpose of such changes is both corrective and perfecting, reflecting errors found and minor code improvements recommended in a typical code inspection. Since code inspection changes are an essential part of the new code development and contain a mixture of purposes, we chose to place code inspection changes in a separate class to be able better to discern the patterns of changes that have a single purpose. As it turned out, the developer perception of change difficulty and the size of code inspection changes were distinct from other types of changes.

After keyword classification, we looked at keywords and designed simple rules to resolve some of the conflicts when keywords of several types are present in one abstract. For

example, the presence of code inspection terms would assign an abstract to the inspection category, independent of the presence of other terms like *new*, or *fix*. The rules were obtained based on our knowledge of the change process (to interpret the meaning of the keyword) and the knowledge obtained from classifying the keywords.

3.4 MR Classification Rules

MR classification rules are applied in sequence and each rule is applied only if no class has already been assigned by a previous rule.

1. Any inspection keyword implies code inspection class. Since the inspection change usually has both perfective and corrective aspects, that is often reflected by appropriate keywords in the abstract. The rule, in effect, ignores such keywords to identify the inspection change.

2. If *fix, bug, error, fixup, fail* are present, the change is classified as corrective. In the keyword classification step, all changes with such keywords in their abstracts had been corrective, indicating that the presence of such keyword strongly increases the probability that the change is corrective. This rule reflects that knowledge.

3. Presence of a keyword determines the type, but if more than one type of keyword is present, the type with most keywords in the abstract prevails, with ties resolved in favor of perfective and then corrective types.

Following examples illustrate the three rules (the actual module, function, and process names have been replaced by three letters). The abstract "Code inspection fixes for module XXX" will be classified as an inspection because of keyword *inspection*. The abstract "Fix multiple YYY problem for new ZZZ process" will be classified as corrective because of keyword *fix*. The abstract "Adding new function to cleanup LLL" will be classified as adaptive because there are two adaptive keywords *add* and *new* and only one perfective keyword *cleanup*.

The MRs abstracts where none of the rules applied, were subjected to classification step 2 (word frequency analysis) and then step 3. There were 33171 MRs of which 56 percent were classified (one of the rules did apply) in the first round and another 32 percent in the second round leaving 12 percent unclassified after the second round. As we later found from developer survey, the unclassified MRs were mostly corrective. One of the possible reasons is that adaptive, perfective, and inspection changes need more explanation, while corrective activity is mostly implied and the use of corrective keywords is not considered necessary to identify the change as a fault fix. The resulting classification is presented in Table 1. It is worth noting that the unclassified MRs represent fewer delta than the classified MRs (12% of MRs were unclassified but less than 10% of delta, added, deleted, or unmodified lines).

A number of change properties are apparent or can be derived from this table.

1. Adaptive changes accounted for 45% of all MRs, followed by corrective changes that account for 34% (46% if we consider the fact that most unclassified changes are corrective).

2. Corrective changes add and delete few lines (they account for 34% to 46% of all changes and only for 18% to 27% of all added and deleted lines).

3. Inspection changes are largest in delta, deleted lines, and unchanged lines.

4. Perfective changes delete most lines per delta (we can see that by looking at the ratio of percentages in the deleted lines row divided by percentages in the delta row).

5. Adaptive changes changed smallest files (ratio of percentages in the unchanged lines row divided by percentages in the MR row). This is not too surprising, since new files are often created to implement new functionality and problems are fixed in larger, mature files.

A more detailed analysis of size and interval relationships is presented in Section 5.1.

4 Validation

The automatic algorithm described above performs classification based solely on textual abstract. To validate the results we collected additional data via the developer survey described in the next section. We also used change size and interval (Section 5.1) to validate the algorithm on a different product (Section 5.2).

4.1 Developer Survey

To calibrate our automatic classification with the developer opinions, we asked a sample of developers to classify their recent MRs. To minimize respondent time and to maximize respondent recall, the survey was done in two stages. In the preliminary stage, a questionnaire containing 10 MRs was given to two developers who were asked to "debug" the survey process and to fine-tune the questions. In the second stage, five other developers were asked to classify 30 of their MRs. The sampling of MRs and the number of

Table 1. Result of the MR Classification Algorithm.

	Corrective	Adaptive	Perfective	Inspection	Unclassified	Total
MR	33.8%	45.0%	3.7%	5.3%	12.0%	33171
delta	22.6%	55.2%	4.3%	8.5%	9.4%	129653
lines added	18.0%	63.2%	3.5%	5.4%	9.8%	2707830
lines deleted	18.0%	55.7%	5.8%	10.8%	9.6%	940321
lines unchanged	27.2%	48.3%	4.5%	10.3%	9.6%	328368903

NOTE: Percentages and totals are presented for MRs, delta, and for lines added, deleted, or unmodified by an MR. In the totals column the number of unmodified lines are added over all changes and is much higher than the total number of lines in any version of the source code.

classes was changed in accordance with the results from the preliminary survey.

4.1.1 Survey Protocol

First we have randomly selected 20 candidate developers who had been working in the organization for more than 5 years and completed most 50 MRs over the last two years. The developer population at the time was stable for the last six years so most developers were not novices.

The subsystem management then selected 8 developers from the list of candidates. The management chose them because they were not on tight deadline projects at the time of the survey. We called the developers to introduce the goals, format, and estimated amount of developer time (less than 30 minutes) required for the survey and asked for their commitment. Only one developer could not participate.

After obtaining developer commitment we sent the description of the survey and the respondents' "bill of rights":

> The researchers guarantee that all data collected will be only reported in statistical summaries or in a blind format where no individual can be identified. If any participants at any time feel that their participation in this study might have negative effects on their performance, they may withdraw with a full guarantee of anonymity.

None of the developers withdrew from the survey. The survey forms are described in Appendix.

4.2 Survey design and results

All of the developers surveyed have completed many more than 30 MRs in the past two years, so we had to sample a subset of the MRs to limit their number to 10 in the preliminary phase and 30 in the secondary phase.

In the first stage we sampled uniformly from each type of MR. The results of the survey (see tables below) indicated almost perfect correspondence between developer and automatic classification. The MRs classified as *other* by the developer were typical perfective MRs, as was indicated in the response comment field and in the subsequent interview. We discovered that perfective changes might be classified both as corrective or adaptive, while all four inspection changes were classified as adaptive.

Table 2. Classification Correspondence Table.

	Automatic Classification				
Dev. Clsfn.	Corr.	Adapt.	Perf.	Insp.	Uncl.
Corrective	6	0	1	0	0
Adaptive	0	5	2	4	1
Other	0	0	1	0	0

NOTE: This table compares labels for 20 MRs between the program doing automatic classification (columns) and developer classification in the preliminary study. Consider the cell with the row labeled "Adaptive" and the column labeled "Perfective". There are 2 MRs in this cell indicating that the developer classified the MRs as "Adaptive" and the program classified them as "Perfective".

To get a full picture in the second stage we also sampled from unclassified MRs and from the perfective and inspection classes. To obtain better discrimination of the perfective and inspection activity we sample with higher probability from from the perfective and inspection classes than from other classes. Otherwise we might have ended with one or no MRs per developer in these two classes.

The survey indicates that automatic classification is much more likely to leave corrective changes unclassified. Hence we assigned all unclassified changes to the type corrective. In the results that follow we assume that all unclassified changes are corrective. This can be considered as the last rule of the automatic classification in Section 3.4.

The overall comparison of developer and automatic classification is in Table 4.

We discussed the two MRs in the row "Other" with the developers. Developers indicated that both represented a perfective activity, however we excluded the two MRs from the further analysis.

Table 3. Comparison Between Automatic and Developer Classification in Follow-up Study of 150 MRs.

	Automatic Classification				
Dev. Clsfn.	Corr.	Adapt.	Perf.	Insp.	Uncl.
Corr.	16	10	4	1	13
Adapt.	7	18	1	0	3
Perf.	6	8	27	9	4
Insp.	1	0	0	21	0
Other	0	0	1	0	0

Table 4. Comparison Between Automatic (columns) and Developer Classification in Both Studies

	Automatic Classification			
Dev. Clsfn.	Corr.	Adapt.	Perf.	Insp.
Corr.	35	10	5	1
Adapt.	11	23	3	4
Perf.	10	8	27	9
Insp.	1	0	0	21
Other	0	0	2	0

More than 61% of the time, both the developer and the program doing the automatic classification put changes in the same class. A widely accepted way to evaluate the agreement of two classifications is Cohen's Kappa (κ) [4] which can be calculated using a statistics package such as SPSS. The Kappa coefficient for Table 4 is above 0.5 indicating moderate agreement [6].

To investigate the structure of the agreement between the automatic and developer classifications we fitted a log-linear model (see [15]) to the counts of the two-way comparison table. The factors included margins of the table as well as coincidence of both categories.

Let m_{ij} be counts in the comparison table, i.e., m_{ij} is the number of MRs placed in category i by automatic classification and category j by developer classification. We modeled $m_{i,j}$ to have Poisson distribution with mean $C + \alpha_i + \beta_j + \sum_{i,j=\mathbf{a,\,c,\,p,\,i}} I(i=j)\gamma_{ij}$, where C is the adjustment for the total number of observations; α_i adjusts for automatic classification margins ($\sum_j m_{ij}$); β_j adjusts for developer classification margins ($\sum_i m_{ij}$); $I(i=j)$ is the indicator function; γ_{ij} represents interactions between the classifications; and indexes $\mathbf{a, c, p, i}$ denote adaptive, corrective, perfective, and inspection classes.

In Table 5 we compare the full model to simpler models. The low residual deviance (RD) of the second model indicates that the model explains the data well. The difference between the deviances of the second and the third models indicates that the extra factor $\gamma_{\mathbf{ii}}$ (that increases the degrees of freedom (DF) by 1) is needed to explain the observed data.

Table 5. Model Comparison

Model formula	DF	RD
$c + \alpha_i + \beta_j + \sum_{i,j=\mathbf{a,\,c,\,p,\,i}} I(i=j)\gamma_{ij}$	5	8.7
$c + \alpha_i + \beta_j + \sum_{i,j=\mathbf{a,\,c,\,p,\,i}} I(i=j)\gamma + \gamma_{\mathbf{ii}}$	7	9.2
$c + \alpha_i + \beta_j + \sum_{i,j=\mathbf{a,\,c,\,p,\,i}} I(i=j)\gamma$	8	26.5

NOTE: Model comparison shows that inspection changes are much more likely to have automatic and developer classifications match than are the other types of changes (estimates in the second model are $\gamma_{\mathbf{ii}} = 1.7, \gamma = 0.65$).

ANOVA table for the second model (Table 6) illustrates the relative importance of different factors.

Table 6. ANOVA Table for the Second Model

Factor	DF	Deviance	Resid. DF	RD
C			15	157
α_i	3	9.4	12	147
β_j	3	16.5	9	131
γ	1	104.4	8	26.5
$\gamma_{\mathbf{ii}}$	1	17.4	7	9.16

NOTE: This table shows that the similarity between two classifications is the most important factor in explaining count distribution, followed by even stronger similarity for the inspection class.

The fact that the coefficient γ is significantly larger than zero shows that there is a significant agreement between the automatic and developer classifications. The fact that the coefficient $\gamma_{\mathbf{ii}}$ is significantly larger than zero shows that inspection changes are easier to identify than other changes. This is not surprising, since for the less frequent types of changes, developers feel the need to identify the purpose, while for the more frequent types of changes the purpose might be implied and only a more detailed technical description of the change is provided.

5 Profiles of Maintenance Types

This section exemplifies some of the possibilities provided by the classification. After validating the classification using the survey, we proceeded to study the size and interval properties of different types of changes. Then we validated the classification properties by applying them to a different software product.

The change interval is important to track the time it takes to resolve problems (especially since we determined which changes are corrective), while change size is strongly related to effort, see, e.g. [1].

5.1 How purpose influences size and interval

Figure 2 compares empirical distribution functions of change size (numbers of added and deleted lines) with change interval for different types of changes. Skewed distribution, large variances, and integer values make more traditional summaries, such as boxplots and probability density plots, less effective. Because of a large sample size, the empirical distribution functions had small variance and could be reliably used to compare different types of maintenance activities.

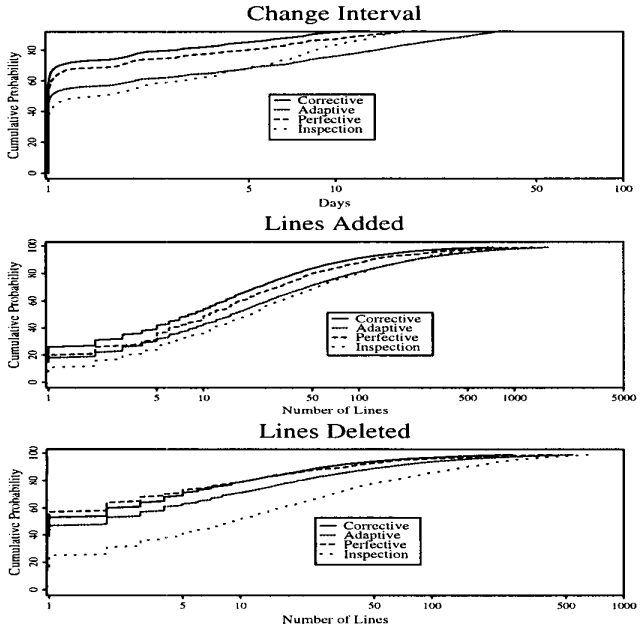

Figure 2. The three plots compare empirical distribution functions of change interval, added lines, and deleted lines for corrective (solid line), adaptive (dotted line), perfective (dashed line), and inspection (long-dashed line) types of changes. Adaptive changes add the most code and take the most time to complete. Inspection changes delete the most code, and corrective changes take the least time to complete.

The empirical distribution functions in Figure 2 are interpreted as follows: the vertical scale defines the observed probability that the value of a quantity is less than the value of the corresponding point on the curve as indicated on the horizontal axis. In particular, the curves to the right or below other curves indicate larger quantities, while curves above or to the left indicate smaller quantities.

The interval comparison shows that corrective changes have the shortest intervals, followed by perfective changes. The distribution functions for inspection and adaptive changes intersect at the 5 day interval and 65th percentile. This shows that the most time consuming 35 percent of adaptive changes took much longer to complete than the the most time consuming 35 percent of inspection changes. On the other hand, the least time consuming 60 percent of inspection changes took longer to complete than corresponding portion of adaptive changes. This is not surprising, since formal inspection is usually done only for changes that add more than 50 lines of code. Even the smallest inspections deal with relatively large and complex changes so implementing the inspection recommendations is rarely a trivial task.

As expected, new code development and inspection changes add most lines, followed by perfective, and then corrective activities. The inspection activities delete much more code than does new code development, which in turn deletes somewhat more than corrective and perfective activities.

All of those conclusions are intuitive and indicate that the classification algorithm did a good job of assigning each change to the correct type of maintenance activity.

All the differences between the distribution functions are significant at the 0.01 level using either the Kruskal-Wallis test or the Smirnov test (see [12]). Traditional ANOVA also showed significant differences, but we believe it is inappropriate because of the undue influence of extreme outliers in highly skewed distributions that we observed. Figure 3 shows that even the logarithm of the number of deleted lines has a highly skewed distribution.

5.2 Variation across products

This section compares the size and interval profiles between changes in different products to validated the classification algorithm on a different software product. We applied the automatic classification algorithm described in Section 3 to a different software product (System B) developed in the same company.

Although System B was developed by different people and in a different organization, both systems have the same type of version control databases and both systems may be packaged as parts of a much larger telecommunications product. We used the keywords obtained in the classification of System A, so there was no manual input to the automatic classification algorithm. System B is slightly bigger

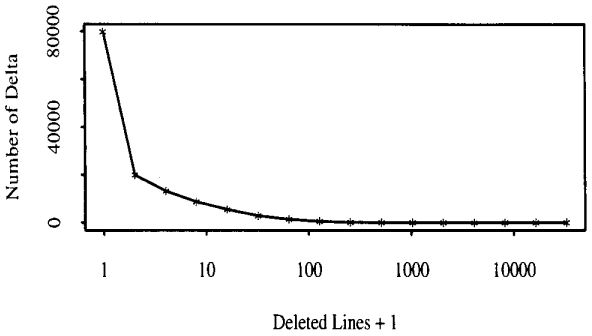

Figure 3. The histogram for the numbers of lines deleted by a delta. Even the logarithm has a highly skewed distribution.

and slightly older than System A and implements different functionality.

Figure 4 checks whether the types of changes are different between the two products in terms of the empirical distribution functions of change size (numbers of added and deleted lines), and change interval.

The plots indicate that the differences between products are much smaller than the differences between types of changes. This suggests that the size and interval characteristics can be used as a signature of change purpose across different software products. However, there are certain differences between the two systems:

1. all types of changes took slightly less time to complete in System B;
2. all types of changes added more lines in System B;
3. corrective changes deleted slightly more lines in System B.

We can not explain the nature of these small differences except that they might be due to the different functionality and developer population in System B.

5.3 Change difficulty

This section illustrates a different application of the change classification by a model relating difficulty of a change to its type. In the survey (see Section 4.1) developers matched purpose with perceived difficulty for 170 changes. To check the relationship between type and difficulty we fitted a log-linear model to the count data in a two-way table: type of changes (corrective, adaptive, perfective, or inspection) versus difficulty of the change (easy, medium, and hard). Table 7 shows that corrective changes are most likely to be rated hard, followed by perfective changes. Most inspection changes are rated as easy.

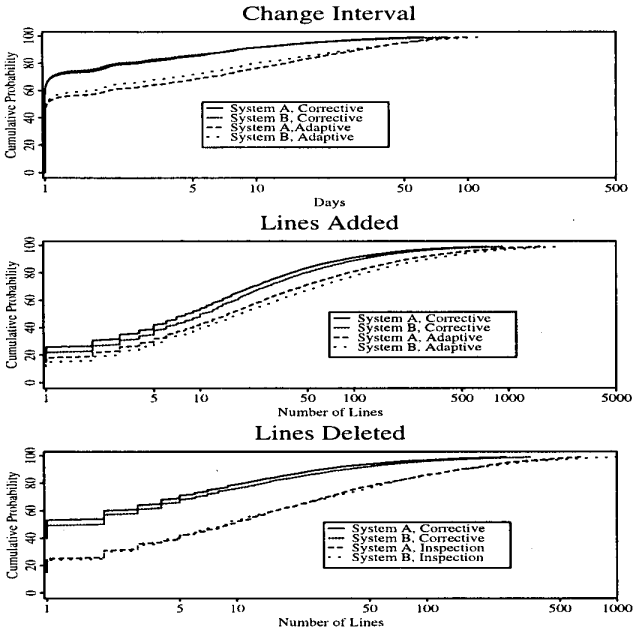

Figure 4. Comparison of two products in terms of empirical distribution functions of change interval and numbers of added or deleted lines for corrective, adaptive, perfective, and inspection changes.

Table 7. Difficulty versus Type of Change.

	Easy	Medium	Hard
corrective	18	21	12
perfective	35	18	1
adaptive	30	8	3
inspection	18	3	1

NOTE: Corrective changes tend to be the hardest while inspection changes are almost always easy.

In the next step we fitted a linear model to find the relationship between difficulty and other properties of the change. Since the difficulty might have been perceived differently by different developers, we included a developer factor among the predictors. To deal with two outliers in the interval (the longest three MRs took 112, 91, and 38 days to complete); we used a logarithmic transformation on the intervals.

We started with the full model:

$$Difficulty = Size + log(Interval + 1) + isAdaptive + isCorrective + isPerfective + isInspection + Developer + Error.$$

Using stepwise regression we arrived at a smaller model:

$$Difficulty = Size + log(Interval + 1) + isCorrective + isPerfective + Developer + Error.$$

Because numbers of delta, numbers of added or deleted lines, and numbers of files touched were strongly correlated with each other, any of those change measures could be used as a change size predictor in the model. We chose to use the number of delta because it is related to the number of files touched (you need at least one delta for each file touched) and to the number of lines (many lines are usually added over several days often resulting in multiple checkins). As expected, the difficulty increased with the numbers of deltas, except for the corrective or perfective changes, which may be small but are still very hard. Not surprisingly, developers had different subjective scales of difficulty. Table 8 gives an analysis of variance (ANOVA) for the full model and Table 9 gives ANOVA for the model selected by stepwise regression. Since R values are so similar, the second model is preferable because it is simpler, having three fewer parameters. We see that the three obvious explanatory variables are size, corrective maintenance, and developer identity. The other two explanatory variables (interval and perfective type), although present in the final model, are not as strong because their effect is not clearly different from zero. This may appear surprising, because interval seems like an obvious indicator of difficulty. However, this is in line with other studies where the change interval (in addition to size) does not appear to help predict change effort [1, 11, 21]. One possible explanation is that the size might account for the difficult adaptive changes, while corrective changes have to be completed in a short time, no matter how difficult they might be.

Table 8. The Full Model with $R = 0.642$

Factor	DF	Sum of Sq.	direction
$Size$	1	7	+
$log(Interval + 1)$	1	0.4	+
isCorrective	1	11.8	+
isAdaptive	1	1e-2	–
isPerfective	1	.5	+
isInspection	1	.3	–
Developer	6	10.3	
Residuals	156	43.6	

Table 9. The Full Model with $R = 0.633$

factor	DF	Sum of Sq.	p-val.	dir.
$Size$	1	7	0	+
$log(Interval + 1)$	1	1.4	0.027	+
isCorrective	1	11.5	0	+
isPerfective	1	0.6	0.12	+
Developer	6	10	0	
Residuals	159	45.8		

6 Summary

We studied a large legacy system to test the hypothesis that historic version control data can be used to determine the purpose of software changes. The study focused on the changes, rather than the source code. To make results applicable to any software product, we assume a model of a minimal VCS so that any real VCS would contain a superset of considered data.

Since the change purpose is not always recorded, and when it is recorded the value is often not reliable, we designed a tool to extract automatically the purpose of a change from the textual description. (The same algorithm can also extract other features). To verify the validity of the classification, we used the developer surveys and we also applied the classification to a different product.

We discovered four identifiable types of changes: adding new functionality, repairing faults, restructuring the code to accommodate future changes, and code inspection rework changes that represent a mixture of corrective and perfective changes. Each has a distinct size and interval profile. The interval for adaptive changes is the longest, followed by inspection changes, with corrective changes being the smallest.

We discovered a strong relationship between the difficulty of a change and its type: corrective changes tend to be the most difficult, while adaptive changes are difficult only if they are large. Inspection changes are perceived as the easiest. Since we were working with large non-Gaussian samples, we used non-parametric statistical methods. The best way to understand size profiles was to compare empir-

ical distribution functions.

In summary, we were able to use data available in a version control system to discover significant quantitative and qualitative information about various aspects of the software development process. To do that we introduced an automatic method of classifying software changes based on their textual descriptions. The resulting classification showed a number of strong relationships between size and type of maintenance activity and the time required to make the change.

7 Conclusions

Our summaries of the version control database can be easily replicated on other software development projects since we use only the basic information available from any version control database: time of change, numbers of added, deleted, and unchanged lines, and textual description of the change.

We believe that software change measurement tools should be built directly into the version control system to summarize fundamental patterns of changes in the database.

We see this work as an infrastructure to answer a number of questions related to effort, interval, and quality of the software. It has been used in work on code fault potential [10] and decay [5]. However, we see a number of other important applications. One of the questions we intend to answer is how perfective maintenance reduces future effort in adaptive and corrective activity.

The textual description field proved to be essential to identify the reason for a change, and we suspect that other properties of the change could be identified using the same field. We therefore recommend that a high quality textual abstract should always be provided, especially since we cannot anticipate what questions may be asked in the future.

Although the purpose of a change could be recorded as an additional field there are at least three important reasons why using textual description is preferable:

1. to reduce developer effort. The description of the change is essential for other purposes, such as informing other developers about the content of the change;
2. to reduce process influences. The development process often specifies deadlines after which no new functionality may be contributed, but in practice there are exceptions that result in relabeling of the true purpose of the change according to process guidelines;
3. to minimize developer bias. Developers opinions may vary and thus influence the labeling of MRs. Using textual description avoids this problem (however it may introduce a different bias due to differences in vocabulary different developers use to describe their MRs.

Acknowledgments

We thank the interview subjects and their management for their support. We also thank Dave Weiss and IEEE TSE reviewers for their extensive comments.

References

[1] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the effectiveness of software tools. In *1999 International Conference on Software Engineering*, Los Angeles, CA, May 1999. ACM Press.

[2] V. R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–737, 1984.

[3] R. Beckwith and G. A. Miller. Implementing a lexical network. *International Journal of Lexicography*, 3(4):302–312, 1990.

[4] J. Cohen. A coefficient of aggreement for nominal scales. *Educational and Psychological Measurement*, 20:37–46, 1960.

[5] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 2000. To appear.

[6] K. El Emam. Benchmarking kappa for software process assessment reliability studies. Technical Report ISERN-98-02, International Software Engineering Network, 1998.

[7] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[8] M. E. Fagan. Advances in software inspections. *IEEE Trans. on Software Engineering*, SE-12(7):744–751, July 1986.

[9] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Software Engineering*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 2000. to appear.

Table 10. MR Information Requested from the Developer for the Calibration Survey.

MR	Type (N/B/I/C/O)	Effort (E/M/H)	Project	MR Opened	Last Delta
wx486218aF			Foo5_2f	1/9/95	1/17/95
Description:	Define a new feature bit mask for LUHN check				
Modules:	uhdr/wx	Files:	WXgl5ac2.G		
Comment:					

[11] T. L. Graves and A. Mockus. Inferring change effort from configuration management databases. *Metrics 98: Fifth International Symposium on Software Metrics*, November 1998.

[12] M. Hollander and D. A. Wolfe. *Nonparametric Statistical Methods*. John Willey, New York, 1973.

[13] P. S. Jacobs. *Text-Based Intelligent Systems*. Lawrence Erlbaum, 1992.

[14] C. F. Kemerer and S. A. Slaughter. Determinants of software maintenance profiles: An empirical investigation. *Software Maintenance: Research and Practice*, 9(4):235–251, 1997.

[15] P. McCullagh and J. A. Nelder. *Generalized Linear Models, 2nd ed.* Chapman and Hall, New York, 1989.

[16] A. K. Midha. Software configuration management for the 21st century. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[17] A. Mockus, S. G. Eick, T. L. Graves, and A. F. Karr. On measurement and analysis of software changes. Technical Report BL0113590-990401-06TM, Lucent Technologies, 1999.

[18] C. J. Van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[19] M. J. Rochkind. The source code control system. *IEEE Trans. on Software Engineering*, 1(4):364–370, 1975.

[20] G. Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Addison-Wesley, Reading, Mass., 1989.

[21] H. P. Siy and A. Mockus. Measuring domain engineering effects on software coding cost. In *Metrics 99: Sixth International Symposium on Software Metrics*, pages 304–311, Boca Raton, Florida, November 1999.

[22] E. B. Swanson. The dimensions of maintenance. In *Proc. 2nd Conf. on Software Engineering*, pages 492–497, San Francisco, 1976.

Appendix

7.0.1 Survey Form

The preliminary survey form asked the developers to limit their time to 20 minutes and presented a list of MRs to be classified. (Table 10 shows the classification information for one MR requested from the developer.) The list was preceded by the following introduction.

> Listed below are 10 MRs that you have worked on during the last two years. We ask you to please classify them according to whether they were (1) new feature development, (2) software fault or "bug" fix, (3) other. You will also be asked to rate the difficulty of carrying out the MR in terms of effort and time relative to your experience and to record a reason for your answer if one occurs to you.
>
> For each MR, please mark one of the types (N = new, B = bug, O = other), and one of the levels of difficulty (E = easy, M = medium, H = hard). You may add a comment at the end if the type is O or if you feel it is necessary.

The second stage survey form began with the following introduction.

> Listed below are 30 MRs that you have worked on during the last two years. We ask you to please classify them according to whether they were (1) new feature development, (2) software fault or "bug" fix, (3) the result of the code inspection, (4) code improvement, restructuring, or cleanup, (5) other. You will also be asked to rate the difficulty of carrying out the MR in terms of effort and time relative to your experience, and to record a reason for your answer if one occurs to you.
>
> For each MR, please mark one of the type options (N = new, B = bug, I = inspection, C = cleanup, O = other), and one of the levels of difficulty (E = easy, M = medium, H = hard), You may add a comment at the end if the type is O or if you feel it is necessary.