Toward Understanding the Rhetoric of Small Source Code Changes

Ranjith Purushothaman and Dewayne E. Perry, *Member*, *IEEE Computer Society*

Abstract—Understanding the impact of software changes has been a challenge since software systems were first developed. With the increasing size and complexity of systems, this problem has become more difficult. There are many ways to identify the impact of changes on the system from the plethora of software artifacts produced during development, maintenance, and evolution. We present the analysis of the software development process using change and defect history data. Specifically, we address the problem of small changes by focusing on the properties of the changes rather than the properties of the code itself. Our study reveals that 1) there is less than 4 percent probability that a one-line change will introduce a fault in the code, 2) nearly 10 percent of all changes made during the maintenance of the software under consideration were one-line changes, 3) nearly 50 percent of the changes were small changes, 4) nearly 40 percent of changes to fix faults resulted in further faults, 5) the phenomena of change differs for additions, deletions, and modifications as well as for the number of lines affected, and 6) deletions of up to 10 lines did not cause faults.

Index Terms—Source code changes, software faults, one-line changes, fault probabilities.

1 Introduction

Change is one of the essential characteristics of software systems [1]. The typical software development life cycle consists of requirements analysis, architecture design, low-level design, coding, testing, delivery, and, finally, maintenance. Beginning with the coding phase and continuing with the maintenance phase, change becomes ubiquitous through the life of the software. Software may need to be changed to fix defects, to change executing logic, to make the processing more efficient, or to introduce new features and enhancements.

Despite its omnipresence, source code change is perhaps the least understood and most complex aspect of the development process. Indeed, source code change is where the essential characteristics of complexity and evolution meet to yield a set of fundamental problems: the implications of changes, the increasing resistance to change—i.e., code decay [1], [5], the risk of changes, etc. Understanding the *rhetoric* of changes is a precondition to effectively creating techniques, methods, processes, and tools to deal with source code changes.

Given that the change size is a significant fault predictor [2], [5], it is not surprising that much of the focus on understanding changes has been on moderate to large-sized changes, while very little attention has been given to understanding small changes. Thus, we focus here on describing the phenomena of, and surrounding, small changes.

- *R. Purushothaman is with the Server Operating Systems Group, Dell Computer Corporation, Round Rock, TX 78682. E-mail: ranjith_purush@dell.com.*
- *D.E. Perry is with the Experimental Software Engineering Lab, Department of Electrial and Computer Engineering, The University of Texas at Austin, Austin, TX 78712. E-mail: perry@ece.utexas.edu.*

Manuscript received 26 Oct. 2004; revised 6 Apr. 2005; accepted 19 Apr. 2005; published online 29 June 2005.
Recommended for acceptance by A. Hassan and R. Holt.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0245-1004.

As mentioned above, managing risk is one of the fundamental problems in building and evolving software systems. How we manage the risk of small changes varies significantly, even within the same company. We may take a strict approach and subject all changes to the same rigorous processes. Or, we may take the view that small changes generally have small effects and use less rigorous processes for these kinds of changes. We may deviate from what we know to be best practices to reduce costs, effort, or elapse times. One such common deviation is not to bother much about one line or other small changes at all. For example, we may skip investigating the implications of small changes on the system architecture, we may not perform code inspections for small changes, we may skip unit and integration testing for them, etc. We do this because our intuition tells us that the risk associated with small changes is also small. However, we all know of cases where one line changes have been disastrous. Weinberg [9] documents an error that cost a company 1.6 billion dollars and was the result of changing a single character in a line of code. Holzmann in his keynote talk at FSE 2002 [22] related similar one-line disasters.

In either case, innocuous or disastrous, we have very little actual data on small changes and their effects to support our decisions. We base our decisions about changes in general and risk in particular on experience, intuition, and anecdotal evidence rather than on a rigorous empirical basis.

One of the factors in managing risk with respect to source code changes is the likelihood of introducing faults. The purpose of this paper, then, is to understand the phenomena of small source code changes in general and the relationship between small changes and faults in particular.

Our approach is different from most other studies that address the issue of software faults because we base the analysis on the properties of the change itself rather than the properties of the code that is being changed [7]. For

example, changes to software can be made by adding new lines, modifying existing lines, or by deleting lines (i.e., the type of the change). Moreover, changes may be made for differing reasons: for example, correcting faults, improving existing code, or adding new features (i.e., the purpose of the change). We expect each of these different types and purposes of change to have different risks of failure.

This study is both descriptive and relational. We describe the phenomena of small changes in the context of all changes. Part of our focus is on one-line changes, part is on small changes (from one to about 10 lines changed), and part is on the entire set of changes (primarily to provide a context for understanding small changes). The research questions we address are as follows:

- How do small changes differ from all the changes?
- What is the relationship of the types and purposes of changes over time?
- What is the relationship between the size of a change and its type and purpose?
- What effect does the size, type, and purpose of a change have on the likelihood of producing a fault?

With respect to this last question, one hypothesis (specific to one-line changes) is that the probability of a one-line change resulting in a fault is small. Another is that the failure probability is higher when the change involves adding new lines of code than either deleting or modifying existing lines.

To answer our questions and test our hypotheses, we use data from the change management and version management systems of a large-scale software project. While historic data from project management systems have been used to analyze the various attributes affecting software development, the use of this data to study the impact of making one-line and other small changes to software has not been done before.

In the next section, we provide a look at past research that has addressed issues related to our analysis. In Section 3, we provide the background for this study, describing the change data and the methods employed for our research. In Section 4, we describe our approach for the analysis of the changed lines and how we prepare the data. In Section 5, we discuss the results of our analysis, summarizing it in Section 6. We discuss threats to validity in Section 7, issues in replication in Section 8, and conclude with recommendations in Section 9. Suggestions for further work are in Section 10.

2 Literature Review

Software maintenance and evolution is the "final phase" of the software life cycle and is frequently viewed as a phase of lesser importance than the initial design and development phases. Quite the contrary, statistical data shows that maintaining 2 to 10 year old software systems demands possibly as high as 40 to 70 percent of the total development effort [15]. We suspect that the number is actually much higher than that. Software maintenance and evolution still remains a difficult process to understand and manage.

Understanding the need for classification of the software changes, Swanson [12] proposed that changes be classified as belonging to three types of maintenance activities. The three types are corrective, adaptive, and perfective. As defined by Swanson, corrective maintenance is performed to correct defects that are uncovered after the software is brought to use. Adaptive maintenance is applied to properly interface with changes in the external processing environment and very often translates into new development and new features. Perfective maintenance is applied to improve nonfunctional characteristics—for example, to eliminate inefficiencies, enhance performance, increase fault tolerance and reliability, or improve maintainability.

Mockus and Votta [3] used the change history from the 5ESS™ switching software project to identify the reasons for software changes. In their analysis, changes were classified as corrective, adaptive, and perfective. They also introduced a fourth type of change classification: changes performed following inspections. Though the changes from inspections were mostly perfective and corrective changes, the number of such changes justified the introduction of a different type of change classification. In any systematic software development environment, code inspections and modifications of code following each inspection are standard procedures. Hence, for our results to be valid in such an environment and since our analysis was also based on the same data, we have retained the "inspection" type of change classification. Our research is based on Mockus and Votta's [3] classification results. Indeed, we used their technique to automatically classify the types of changes and provide that change data for our study.

In his analysis, Hatton [17] related the defect frequency to file size. He stated that contrary to conventional wisdom that smaller components contain fewer faults, medium-sized components are proportionally more reliable than small or large ones.

Analysts use both product measures such as the number of lines of code and process measures such as those obtained from the change history [10]. In their study looking for factors to predict fault incidence, Graves et al. [13] stated that, in general, process measures based on change history are more useful in predicting fault rates than product measures. They gave an example of how a process metric such as the number of times the code has already been changed is a better indication of how many faults it will contain than its length (which is a product measure). Their study concluded that a module's expected number of faults is proportional to the number of times it has been changed.

Mockus and Weiss [7] studied the relation between the size of the change and probability of error and found that the failure probability increases with the number of changes, the number of lines of code added, and the number of subsystems touched. They also concluded that the probability of error is much higher for new development as compared to defect fixes because the change size associated with defect fixes tend to be much smaller in size. Dunsmore and Gannon [14] state that there is statistical evidence (Spearman $\rho = 0.56$ with $\alpha = .05$)[1] that shows a

1. Spearman's ρ is a correlation coefficient for data in rank form; 0.56 represents a medium effect.

direct relationship between the amount of program changes and the error occurrences.

In the analysis done by Leszak et al. [2], the authors conclude that large changes to existing code are fault prone and provide statistical data to support their claim. They go a step further to propose that changes of more than 25 percent of existing code should be avoided and recommend recoding instead of modification. Basili and Perricone [18] categorize software modules based on their size (lines of code) and then checked for the errors at the module level. An interesting observation from their research was that, of the modules found to contain errors, 49 percent were categorized as modified and 51 percent as new modules.

Our primary contribution in this empirical research is an initial descriptive and relational study of small changes. As shown from our related research discussion above, we are the first to study this phenomenon. Another unique aspect of our research is that we have used a combination of product measures such as the lines of code and process measures such as the change history (change dependency) to analyze the data. In doing so, we have tried to gain the advantages of both measures while removing any bias associated with each of them.

While several papers discuss the classification of changes based on its purpose (corrective, adaptive, preventive) there is virtually no discussion based on the type of change: software can be changed by adding lines, deleting lines, or by modifying existing lines. As a byproduct of our analyses, we have provided useful information that gives some insight into the impact that the type of change has on the software evolution process.

The 5ESS™ change and version history data that we use here has been used for various research purposes, such as for inferring change effort from configuration management databases [4], studying the impact of parallel changes in large-scale software development projects [16], analyzing the challenges in evolving a large-scale software product [6], identifying the reasons for software changes [3], and predicting fault incidence [13], to name a few. The wide range of studies that have used this particular change history data ensures good content validity for the results of the analyses based on this data.

3 The Context of the Study

The context of this study is Lucent Technologies' 5ESS™ telephone switch [22]. 5ESS itself is comprised of approximately 100,000,000 lines in C, C++, and a number of domain specific languages, plus another 100,000,000 lines of header and make files. The system is organized into 50 subsystems, which are in turn organized into a total of 5,000 modules[2] (where a module is a directory of source code files). Each release consists of approximately 20,000,000 lines of code.

For this study, we use the same 5ESS subsystem used in the Code Decay Project [5]—that is, the complete change and version management history from the first 15 years of that subsystem. This subsystem was built at a single development site within a single organization, the structure of which has varied over the years reaching a peak of 200 developers and eventually decreasing to the current 50 developers. The members of technical staff (MTS) who developed 5ESS typically have MS degrees. While their degrees range over a wide range, most are from scientific areas and all are subject to initial training courses in telephony.

To coordinate this very large project, 5ESS has well-defined processes available online with a process management organization responsible for their creation, evolution, and improvement. The processes are described informally but in a highly structured manner and occupy over 100 megabytes of disc space and are accessed more than 500 times a day by the developers [23]. Some of these processes represent specific cases where small-sized changes are exempt from the standard processes. For example, the architecture process is such a case: Small changes are not given the full architectural review.[3] In other cases, they have recognized that the size of features (the basic unit of development) is important enough to differentiate the design process according to size [24].

In this section, we describe the change process in the 5ESS software development project and also give an introduction to the product subsystem that we use for our analysis.

3.1 The 5ESS Change Process

In 5ESS, a *Feature* is the fundamental unit of system functionality to be added to the system. Each Feature is implemented by a set of Initial Modification Requests (IMRs) where each IMR represents a logical problem to be solved and as such are the basic unit of development. Features range in size from those requiring only one developer to those requiring development teams in the hundreds, and may be either software only or a combination of software and hardware. Features then may require anywhere from one IMR to hundreds of IMRs.

Each IMR is implemented by a set of functionally distinct Modification Requests (MRs) where each MR represents a logical part of an IMR's solution (and a developer may further subdivide a solution into multiple MRs if they are seen as multiple logical sets of changes). Multiple developers may well be involved in solving an IMR; however, only one developer is assigned to (i.e., "owns") a specific MR. Of course, a developer may be responsible for more than one MR and may easily work on more than one IMR. The solutions represented by MRs (to the problems described by the IMRs) are reified in terms of changes to source code by changes to one or more files. Thus, there are three logical units of change: Feature, IMR, and MR.

The change history [3], [5], [7] of the system is maintained using the IMR Tracking System (IMRTS), the Extended Change Management System (ECMS [25]) for initiating MRs and tracking changes, and the Sources Code

2. Note that where earlier we used the term *module* to mean a generic component, in 5ESS the term *module* has a specific technical meaning of a component that is of larger granularity than that used earlier in the paper.

3. One of the authors was a member of the team at AT&T (before the split between Lucent and AT&T) defining the requirements for best in practice processes in which it was required that all changes regardless of size be given a full review. Those responsible for the 5ESS architecture process argued that the recommendation ought to be changed to reflect their practice of not reviewing small changes to the system.

Control System (SCCS [26]) for managing different versions of the files. Features and their associated IMRs are managed by IMRTS and include information about the associated MRs. ECMS records information about each MR and its relationship to SCCS's versions. Each IMR has an extensive set of information (89 fields in all). Each MR includes such data as the parent IMR, dates, the affected files, and text describing the changes and their rationale.

The actual source code changes themselves are maintained by SCCS in the form of one or more *deltas* (the physical unit of change) depending on the way the changes are committed by the developer. Each delta provides information on the attributes of the change: the parent MR, lines added, lines deleted, lines unchanged, login of the developer, and the time and date of the change.

While it is possible to make all changes that are required to be made to a file by an MR in a single delta, developers often perform multiple deltas on a single file for an MR. Multiple deltas are common because of the amount of parallel changes in building a new release [16]. Because of the size of the system together with the fact that, as the system ages, an increasingly large portion of the system remains unchanged [27], there has been a shift away from code ownership in evolving the 5ESS system. The unit of development is now a Feature (on average about 500 new Features per release) and these proceed in parallel with multiple MRs affecting individual files. Since SCCS does not support parallel versions, quasi-parallel versions are represented as a series of sequential but interleaved deltas. A series of deltas for a single MR is thus an artifact of parallel changes in a version management system that does not support parallel versions. It is for this reason that we consider an MR to represent one logically physical change per file.

3.2 5ESS Change Data

The 5ESS™ subsystem we use for this study contains 4,550 files that have a total of nearly 2 million lines of code. Over the last decade, this subsystem had 31,884 MRs that changed nearly 4,293 files. So, nearly *95 percent of all files were modified* after first release of the product.

Change to software can be introduced and interpreted in many ways. However, our definition of change to software is driven by the historic data that we used for this analysis: A change is *any alteration to the software recorded in the change history database* [5]. In accordance with this definition, in our analysis the following were considered to be changes:

- one or more modifications to single/multiple lines,
- one or more new statements inserted between existing lines,
- one or more lines deleted, and
- a modification to a single/multiple lines accompanied by insertion or/and deletion of one or more lines.

Since we are looking at the phenomena of small changes specifically in this study, the definition of one-line changes is of critical importance since there are several ways in which we might view them. For example, we could consider one line changes with respect to a delta, a file, or an MR. But, as we have already discussed, multiple deltas are an artifact of the tools used that forces interleaving of deltas for an MR and, hence, we view the definition in terms of deltas as unsatisfactory. Similarly, one could make an argument for viewing the one-line changes on a per file basis. There is some merit in this choice, but it ignores the fact that an MR represents a set of related changes for one specific solution. Hence, we decided that an MR, a unit of logical change, is the most conservative choice. The following kinds of changes, then, would qualify as a one-line change for this study when an MR consists of either:

- one or more modifications to a single line,
- one or more lines replaced by a single line (i.e., multiple lines deleted, one line added),
- one new statement inserted between existing lines, or
- one line deleted.

Previous studies such as [14] do not consider deletion of lines as a change. However, from preliminary analysis, we found that lines were deleted for fixing bugs as well as making modifications. Moreover, in the SCCS system, a line modification is tracked as a line deleted and a line added. Hence, in our research, we have analyzed the impact of deleting lines of code on the software development process.

4 Approach

In this section, we document the steps we took to obtain useful information from our project database. We first discuss the preparation of the data for the analysis and then explain some of the categories into which the data is classified. The final stage of the analysis identifies the logical and physical dependencies that exist between files and MRs.

4.1 Data Preparation

The change history database provides us with a large amount of information. Since our research focuses on analyzing one-line changes and changes that were dependent on other changes, one of the most important aspects of the project was to derive relevant information from this data pool. As mentioned above, multiple deltas per MR are a physical artifact resulting from the version management system SCCS lacking support for parallel changes in terms of logically parallel versions and are merged when the changes are completed. The logically parallel versions are represented as a sequence of interleaved deltas.

In the change process hierarchy, an MR is the lowest logical level of change. Hence, if the MR was created to fix a defect, all the modifications that are required by an MR would have to be implemented to fix the bug. Hence, we were interested in change information for each effected file at the MR level not at the delta level. For example, in Table 1, the MR *o401472pQ* changes two files. Note that the file *oaMID213* is changed in two steps. In one of the deltas, it modifies only one line. However, this cannot be considered to be a one-line change since for the complete change, the MR changed three lines of the file. Thus, we have had to process the delta data to merge multiple deltas into one logical change to each file in an MR.

With nearly 32,000 MRs that modified nearly 4,300 files in the subsystem under investigation, the aggregation of

TABLE 1
Delta Relation Snapshot

DELTA relation				
MR	**FILE**	**Add**	**Delete**	**Date**
Oa101472pQ	oaMID213	2	2	9/3/1986
Oa101472pQ	oaMID213	1	1	9/3/1986
Oa101472pQ	oaMID90	6	0	9/3/1986
Oa101472pQ	oaMID90	0	2	9/3/1986

the changes made to each file at the MR level gave us 72,258 change records for analysis. We note that with this number of samples, what appears to be significantly different in the distributions shown below is likely to be significantly different statistically. However, for completeness' sake, we discuss the results of χ^2 analyses (and provide the complete analyses in the Appendix) to show the extent to which the differences are indeed significant.

4.2 Data Classification

Change data can be classified based on the purpose of the change and also based on how the change was implemented. The classification of the MRs based on the change purpose was derived from the work by Mockus and Votta [3] (and our data classified with help from Mockus). They classified MRs based on the keywords in the textual abstract of the change. For example, if keywords like "fix," "bug," "error," and "fail" were present, the change was classified as corrective. In Table 2, we provide a summary of the change information classified based on its purpose. The naming convention is similar to the work done in their original paper. Perfective changes are typically made to improve performance, make a piece of code more maintainable, or generally improve the quality of the code. Adaptive changes are those in which new features and functionality are added to the system.

However, there were numerous instances when changes made could not be classified clearly. For example, certain changes were classified as "IC" since the textual abstract had keywords that suggested changes from inspection (I) as well as corrective changes (C). Though this level of information provides for better exploration and understanding, in order to maintain simplicity, we made the following assumptions:

- Changes with multiple "N" were classified as "N."
- Changes with multiple "C" were classified as "C."
- Changes containing at least one "I" were classified as "I."

Changes which had "B" and "N" combinations were left as "Unclassified" since we did not want to corrupt the data. Classification of these as either a corrective or adaptive change would have introduced validity issues in the analysis. Based on the above rules, we were able to classify nearly 98 percent of all the MR into corrective, perfective, adaptive, or inspection changes.

TABLE 2
Change Purpose

ID	Change Purpose	Explanation
B	Corrective	Fix defects
C	Perfective	Enhance performance
N	Adaptive	New development
I	Inspection	Following inspection

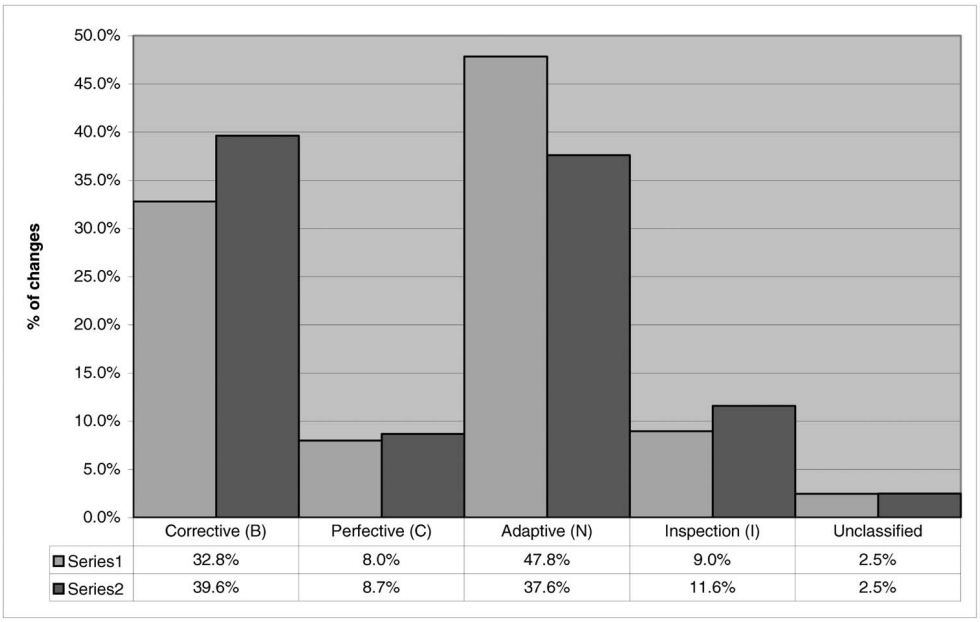
Fig. 1 compares the distribution of one-line changes against the total set of changes classified according to purpose. For the complete set of changes, the largest set of changes is due to adaptation. This is not surprising when considering the fact that each release of the system incorporates a significant number of new features. The second largest set of changes is due to corrections. Again, this is not surprising considering the official development and evolution processes begin tracking defects as soon as testing begins. Perfective and inspection sets of changes are respectively relatively small. Conversely, for one-line changes, corrective changes are the most frequent while adaptive are slightly less frequent.

The distributions shown in Fig. 1 of one-line changes versus all the changes are significant. In the χ^2 analysis shown in Table 4 (in the Appendix), the value of $\chi^2 = 234.995$ with 4 degrees of freedom and with $p \leq .001$. One-line corrective changes are significantly higher than expected while one-line adaptive changes are significantly lower than expected. Thus, compared to the complete set of changes, one-line changes are used to fix faults significantly more often but are used for adaptive purposes significantly less often.

Another way to classify changes is on the basis of the implementation method into insertion, deletion, or modification. But, the SCCS system maintains records of only the number of lines inserted or deleted for the change and not the type of change. Modifications to the existing lines are tracked as old lines being replaced by new lines (delete and insert). However, for every changed file SCCS maintains an SCCS file that relates the MR to the insertions and deletions made to the actual file. Scripts[4] were used to parse these SCCS files and categorize the changes made by the MR into inserts, deletes or modifications. Table 3 lists different types of changes based on their implementation method. Inserts and deletes are straightforward; modifications are accomplished by a combination of delete and inserts (An analog to an SCCS file is that produced by the `diff` command in UNIX™). The scripts parse and analyze the records into the various classifications listed in Table 3.

Fig. 2 compares the distribution of one-line changes against all the changes classified by change type—i.e., by the means of change implementation. Given that, for all

4. The scripts made available for our research were those used in the Code Decay project at Bell Labs (Perry was a member of that project) and were used for a variety of studies in that project as well as other analysis and visualization (see, for example, [32]) studies in the Software Production Research Project, Bell Labs, Lucent Technologies. In this case, the scripts were provided by Tom Ball, now of Microsoft Research.



	Corrective (B)	Perfective (C)	Adaptive (N)	Inspection (I)	Unclassified
all	23687	5764	34557	6473	1777
one-line	2259	495	2144	661	142

Fig. 1. Distribution of changes based on purpose (series 1: all changes; series 2: one-line changes).

changes, the most frequent purpose of changes to the system is adaptation, it is not surprising that the most frequent means of implementing changes is by inserting lines. Change and the various combinations are about equal in distribution. Only a very small set of changes were accomplished solely by deleting lines. This low percentage very likely stems from two reasons.[5] First, actual releases are configured from a large set of features and customized for specific uses, so there is little motivation to remove functionality. Second, very little functionality is removed for fear of breaking the system.

For one-line changes, the distibution is significantly different from the entire set of changes. In the χ^2 analysis shown in Table 5 (see the Appendix), the value of $\chi^2 = 1615.098$ with 2 degrees of freedom (because we leave out combinations from the comparison) and with p significantly less than .001. Modifications are used significantly more often in one-line changes than in the total set of changes, while inserts are used significantly less often. Lines are deleted significantly more often in one-line changes. The way one-line changes are done, then, is significantly different that the total set of changes.

4.3 Identifying Change Dependencies

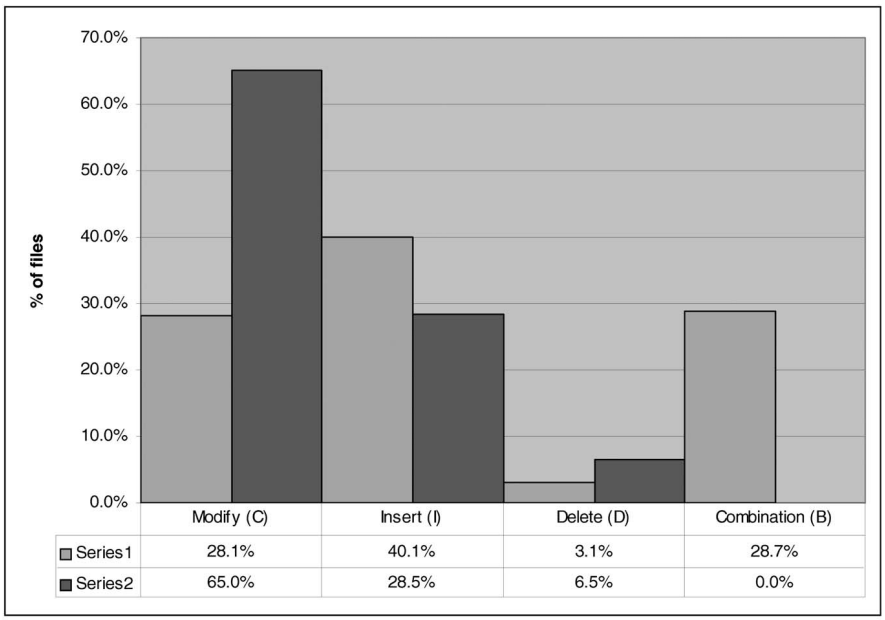
Our primary concern was in isolating those changes that resulted in errors. To do so, we identified those changes that were *dependencies—changes to lines of code that were changed by an earlier MR*. If the latter change was a bug fix, our assumption was that the original change was in error. There are two ways in which the original change might be in error: an error by commission—i.e., the change was incorrect in that it introduced an error with the change; an error by omission—i.e., the change was incorrect in that it didn't make sufficient changes to solve the problem.

The one argument against the validity of this assumption would be that the latter change might have fixed a defect that was introduced before the original change was made. However, in the absence of prima facie evidence to support either case and since preliminary analysis of some sample data did not support the challenging argument, we ruled out this possibility. In this report, we will refer to those files in which changes were made to those lines that were changed earlier by another MR as *dependent files*.

TABLE 3
Change Type

ID	Change Type	Description
C	Modify	Change existing lines
I	Insert	Add new lines
D	Delete	Delete existing lines
IC	Insert/Modify	Inserts and modifies lines
ID	Insert/Delete	Inserts and deletes lines
DC	Delete/Modify	Deletes and modifies lines
DIC	All of the above	Inserts, deletes and modifies lines

5. This is based on anecdotal evidence gathered by one of the authors when working with 5ESS developers.



	Modify (C)	Insert (I)	Delete (D)	Combination (B)
all	20286	28974	2232	20766
one line	3708	1624	369	

Fig. 2. Distribution of changes based on type (series 1: all changes; series 2: one-line changes).

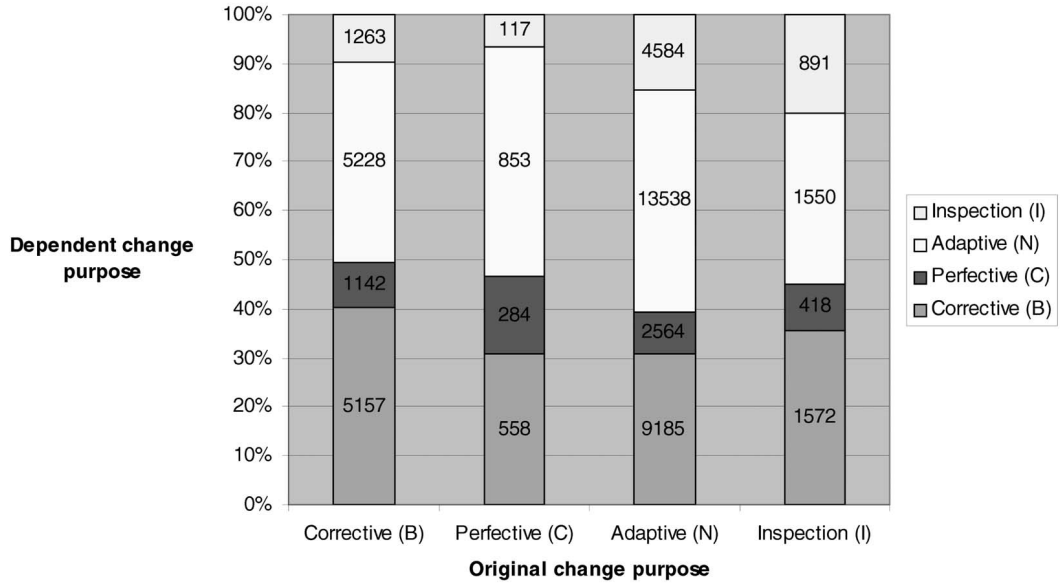


Fig. 3. Distribution of change classification on dependent files.

The dependency, as we have defined earlier, may have existed due to bug fixes (corrective), enhancements (perfective), changes from inspection, or new development (adaptive). We studied 2,530 files in the subsystem and found them to have dependent changes. That is nearly 55 percent of all files in the subsystem and nearly 60 percent of all changed files. So, *in nearly 60 percent of cases, lines that are changed were changed again*. This kind of information can be very useful to the understanding of the maintenance phase of a software project. We had 51,478 dependent change records (of all size changes) and this data was the core of our analysis.

In Fig. 3, we show the distribution of change classifications of the dependent files across the original files. The horizontal axis shows the types of changes made to the dependent files originally. In the vertical axis, we distribute the new changes based on their classification based on the purpose type. We note first that the bulk of dependent changes to all categories were adaptive and corrective. We note second that the largest percentage of bug fixes was made to code that was already changed by an earlier MR to

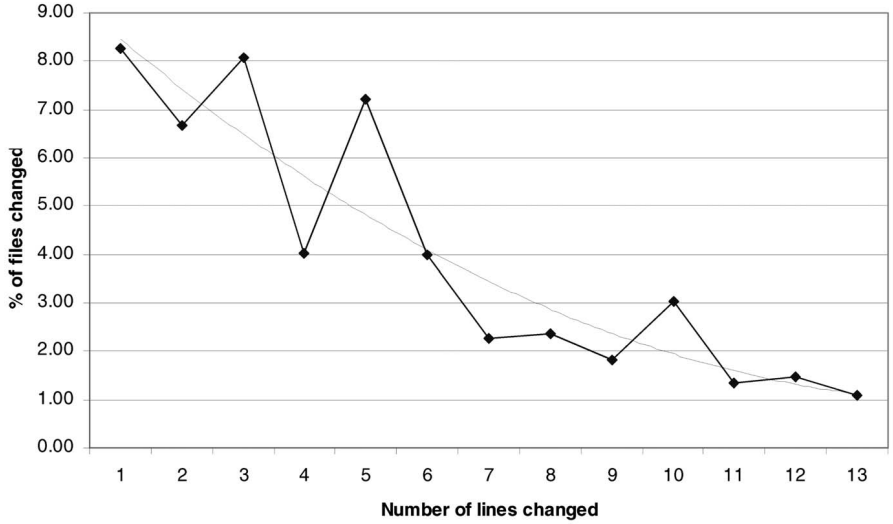


Fig. 4. Distribution of small changes.

fix bugs—that is *roughly 40 percent of changes made to fix errors introduced more errors*(as shown by the fact that 40 percent of dependent changes made to corrective changes were corrective).

It is also interesting to note that nearly 40 percent of all the dependent changes were of the adaptive type and most perfective changes were made to lines that were previously changed for the same reason, i.e., enhancing performance or removing inefficiencies.

As one would expect, the structure of the distribution is significant—i.e., the two populations are not independent. In the χ^2 analysis shown in Table 6 (see the Appendix), the value of $\chi^2 = 856.36$ with 9 degrees of freedom and with $p \leq .001$. Interestingly, the number of dependent corrective changes made to corrective changes is significantly higher than would be expected, while it is significantly lower than expected for adaptive changes. The number of dependent perfective changes is significantly higher than expected for original perfective changes. Adaptive dependent changes were significantly higher than expected for adaptive changes, but significantly lower than expected for corrective changes. All dependent inspection changes were significant: lower than expected for corrective and perfective changes, higher that expected for adaptive and inspection changes.

The original corrective changes had significantly more than expected dependent corrective changes but fewer dependent adaptive and inspection changes. The original set of perfective changes had significantly more dependent perfective changes and fewer inspection changes than expected, while the original set of adaptive changes had significantly fewer dependent corrective changes and significantly more dependent adaptive and inspection changes. Original inspection changes had significantly fewer dependent adaptive changes and significantly more dependent inspection changes.

While not necessarily the largest magnitude in each case, each change purpose had more dependent changes of the same purpose than expected.

5 Analysis

The analysis of the data proceeds in several steps. We begin with an investigation of the software project based on the change size. We then look at the problem of changes introducing defects and complete our analysis by considering the issues surrounding change process metrics.

5.1 Change size

Change size is an effective way to estimate the change effort in a software development project. From our analysis, we were able to derive meaningful information that gives a measure of the number of lines that are changed as part of an MR. Fig. 3 shows the distribution of the changed files based on the number of lines per MR that were changed. The vertical axis shows the percentage of changed files that changed the number of lines specified on the horizontal axis. Note that one-line changes (i.e., one line in one MR) are of necessity changes to only one file. MRs with multiple lines changed may (and often do) affect more than one file. It is for this reason that it is useful to relate the number of lines changed per MR to the number of files changed.

Fig. 4 plots the number of lines changed (horizontal axis) against the percentage of files changed with that number of lines (vertical axis), we can see that *nearly 10 percent of files changed involved changing only a single line of code*. Since the data fluctuated slightly, we did a second degree polynomial regression analysis of the data as shown by the regression line in the figure. From the regression line obtained, we can see that percentage of effected files reduces as the size of the change increases. Nearly *50 percent of all changed files involved changing 10 lines of code* or less.

So, though the effort for changing one-line of code is generally smaller, the magnitude of these changes is very large in the software evolution process. However, it has been found that developers tend to give less priority to smaller changes and especially one-line changes. To illustrate further, Fig. 5 shows the distribution of all the changed files in the subsystem under study across their change sizes. From this figure, we note that *nearly 95 percent of all changes were those that changed less than 50 lines of code*.

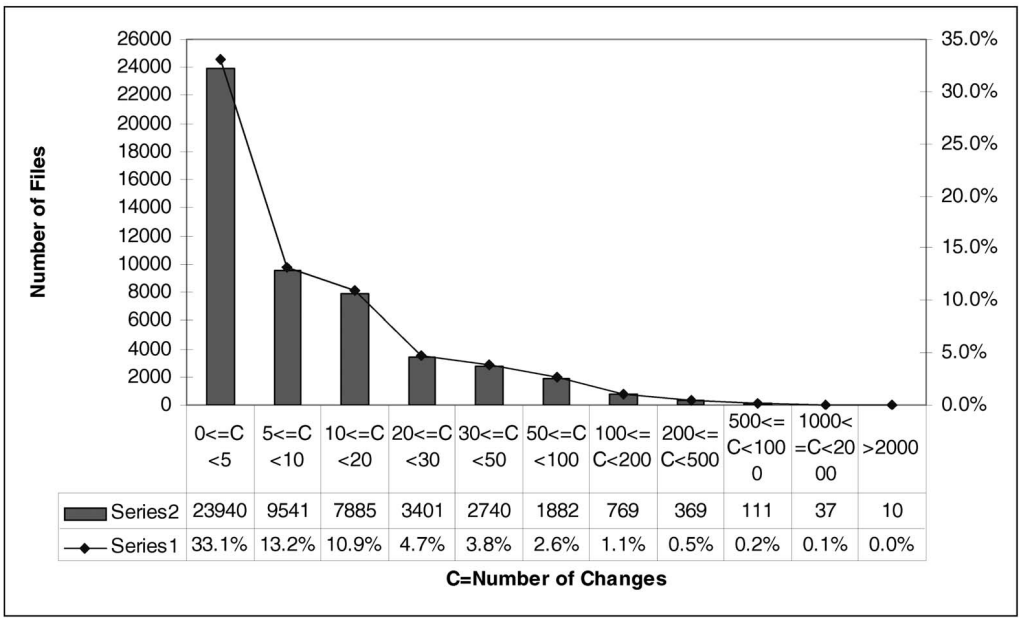


Fig. 5. Change size distribution across files.

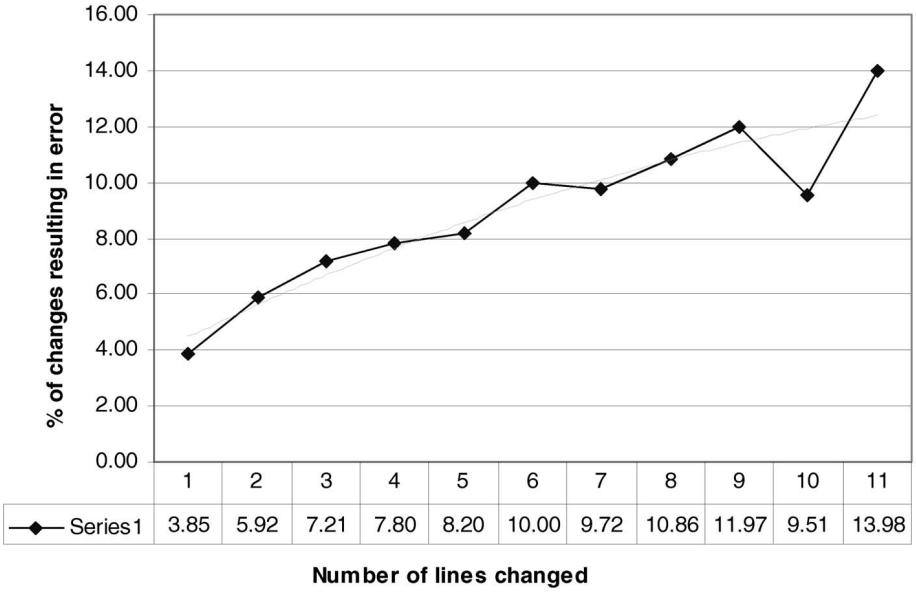


Fig. 6. Errors introduced by change.

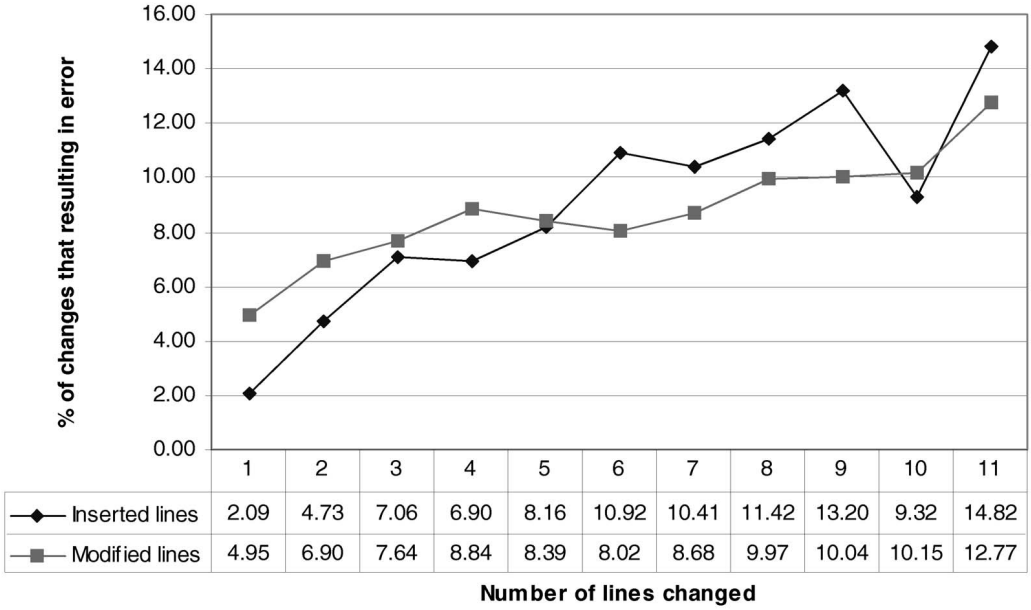
5.2 Erroneous changes

We next analyze those changes that resulted in errors. In Fig. 6, we present the data for erroneous changes that affected less than 10 lines of code. The vertical axis gives the percentage of changes that resulted in error out of the total changes that affected the number of lines specified in the horizontal axis. The data was derived from the change file dependencies that we had defined in an earlier section of this paper. This analysis also answers a very important question: What percentage of one-line changes result in error? *Less than 4 percent of one-line changes result in errors* (see Fig. 6).

It may also be noted that the changes tend to be more erroneous as the number of lines changed increases. One possible explanation to this behavior can be that as the number of lines that are changed increases, there are more avenues provided for the developer to make mistakes. These increased opportunities to introduce errors are likely due to an increase in the number of possible interactions with other lines of code.

We mentioned earlier the classification of changes based on their type into changes by insertion, deletion, and modification. We thought it would be a useful metric to analyze the distribution of erroneous changes based on the type of change. Fig. 7 shows the results of this analysis. Changes made by deletion of lines have been excluded since our analysis *did not produce any credible evidence that deletion of less than 10 lines of code resulted in errors*.

From Fig. 7, we note that while the probability that an insertion of a single line might introduce an error is 2 percent, there is nearly a 5 percent chance that a one-line modification will cause an error. It can also be seen that while modified lines cause more errors when less than



	1	2	3	4	5	6	7	8	9	10	11
Inserted lines	2.09	4.73	7.06	6.90	8.16	10.92	10.41	11.42	13.20	9.32	14.82
Modified lines	4.95	6.90	7.64	8.84	8.39	8.02	8.68	9.97	10.04	10.15	12.77

Fig. 7. Erroneous changes by type of change.

5 lines are changed; inserted new lines introduce more errors with larger change sizes.

To emphasize this behavior, in Fig. 8, we have shown the distribution of the probability of error introduced by change over a wider range of change sizes. It may be noted that there is nearly a 50 percent chance of at least one error being introduced if more than 500 lines of code are changed. The trend of the lines for change implemented by lines inserted and modified clearly shows that insertion of new lines generates a lot more errors when the change size is higher. One plausible explanation for this may be that developers tend to be more cautious when existing code has to be modified than when new development is done.

5.3 Change Process Metrics

How are the types of change related to change classifications? In Fig. 9, the vertical axis categorizes changes based on their purpose and the horizontal axis classifies changes based on how the change was implemented. As expected, the largest number of lines was inserted for adaptive changes since new development involves addition of new lines of code. Modifications were made to existing lines of code equally for both adaptive and corrective changes.

We can see that the Fig. 9 holds no surprises except maybe that deletion of lines occurred pretty much uniformly for adaptive, corrective, and perfective changes. Note, however, that there is a higher percentage of deleted lines than modified, inserted and combined in perfective evolution. In the χ^2 analysis shown in Table 7 (see the Appendix), the value of $\chi^2 = 5,365.58$ with 12 degrees of freedom and p significantly less than .0001. While the overall distribution is significant, the unclassified changes are fairly independent—i.e., the observed and expected are not significantly different.

Among the significant facts about the structure were the following. Corrective changes were modified more than would be expected while a combinations were applied less often then expected. Perfective changes had 30 percent more insertions and 400 percent more deletions than were expected and half the combinations and a quarter the modifications. Adaptive changes had fewer modifications and half the deletions expected but more combinations. Everything about inspection changes: ~75 percent more modifications than expected, half the insertions expected, more deletions, and fewer combinations.

Thus, there are the following significant tendencies: to make more modifications to corrective and inspection changes and fewer to perfective and adaptive changes; to insert approximately the expected number of changes for corrective and adaptive changes, but insert more lines in perfective but fewer lines in inspection changes; to delete more lines than expected for perfective and inspection changes, fewer for perfective; to apply fewer combinations to corrective, perfective, and inspection changes, and more to adaptive.

Fig. 10 continues this discussion but restricts the change data to only one-line changes. The similarity of the data distribution in the two figures show that the behavior of one-line changes at least in regard to their distribution among the change types is representative of the behavior of changes irrespective of the size of the change. The only notable difference between the data in Fig. 9 and Fig. 10 is in the case when new single lines are inserted—less than 2.5 percent of one-line insertions were for perfective changes compared to nearly 10 percent of insertions towards perfective changes when all change sizes were considered.

In the χ^2 analysis shown in Table 8 (see the Appendix), the value of $\chi^2 = 458.54$ with 8 degrees of freedom and $p = .001$ showing a significant distribution. The unclassified one-line changes made were fairly independent (i.e., the expected and observed values were not significantly different. Modifications were close to expected for corrective, perfective, and adaptive one-line changes, but more than expected for inspection changes. Insertions were about half of what was expected for perfective and inspections changes, but about 30 percent more than expected for adaptive changes.

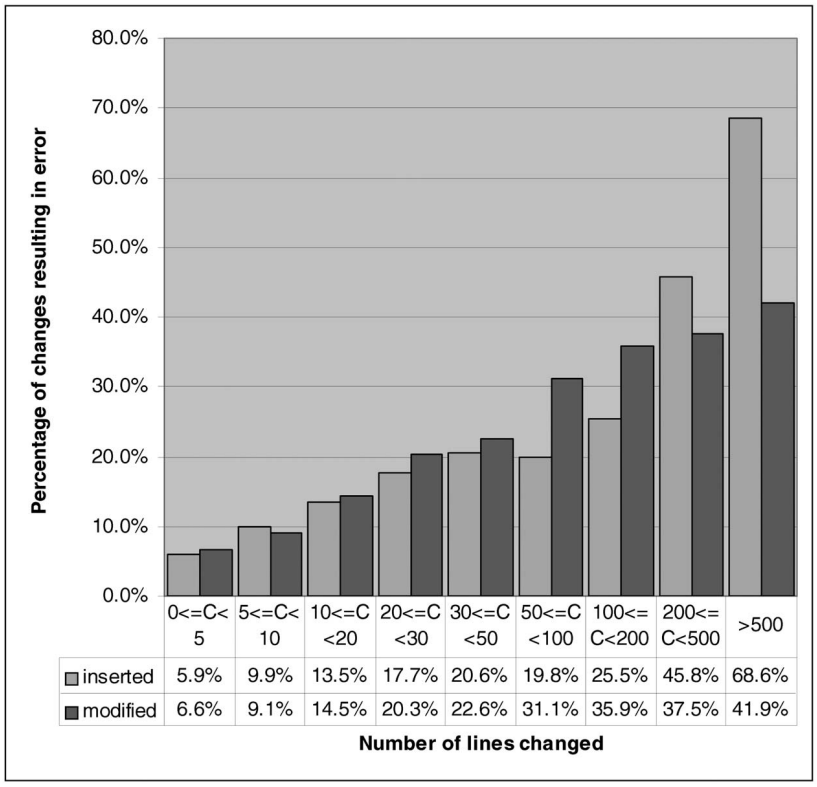


Fig. 8. Erroneous changes versus change size.

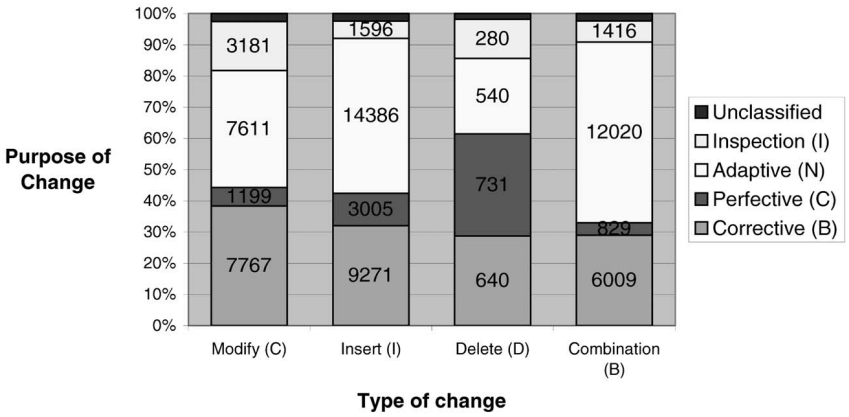


Fig. 9. Relation between change classification and change type.

Deletions were significantly more for perfective (400 percent) and adaptive (300 percent). There was nothing significant about the type of changes to corrective changes and the types of changes were pretty much as expected. Fewer lines were inserted than expected and more lines deleted than expected in perfective changes, while there were more lines inserted and deleted than expected in adaptive changes. For inspection changes, more lines were modified and fewer inserted than expected.

6 Analysis Summary

We have found that the probability that a one-line change would introduce one error is less than 4 percent. This result supports the typical risk strategy for one-line changes and puts a bound on our search for catastrophic changes. Interestingly, this result is very surprising considering the anecdotal claim of a software guru that "*one-line changes are erroneous 50 percent of the time.*" This large deviation may be attributed to the structured programming practices, and development and evolution processes involving code inspections and walkthroughs that were practiced for the development of the project under study. Earlier research [9] shows that without proper code inspection procedures in place, there is a very high possibility that one-line changes could result in error.

We have also provided key insights that can be very useful for better understanding the software development and evolution process. In summary, some of the more interesting observations that we made during our analysis include:

- Nearly 95 percent of all files in the software project were maintained at one time or another. If the

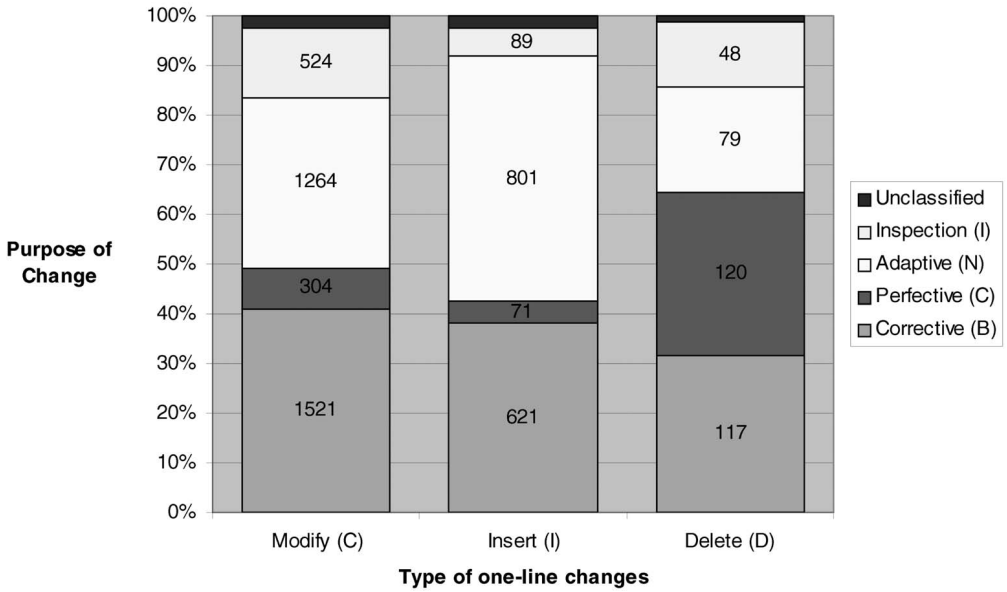


Fig. 10. Relation between various change types for one-line changes.

common header and constants files are excluded from the project scope, we can conclude that nearly 100 percent of files were modified at some point in time after the initial release of the software product.

- Nearly 40 percent of changes that were made to fix defects introduced one or more other defects in the software.
- Nearly 10 percent of changes involved changing only a single line of code, nearly 50 percent of all changes involved changing 10 or fewer lines of code, and nearly 95 percent of all changes were those that changed fewer than 50 lines of code.
- Less than 4 percent of one-line changes result in error.
- The probability that the insertion of a single line might introduce a defect is 2 percent; there is nearly a 5 percent chance that a one-line modification will cause a defect. There is nearly a 50 percent chance of at least one defect being introduced if more than 500 lines of code are changed.
- Less than 2.5 percent of one-line insertions were for perfective changes, compared to nearly 10 percent of insertions toward perfective changes when all change sizes were considered.
- The maximum number of changes was made for adaptive purposes, and most changes were made by inserting new lines of code.
- There is no credible evidence that deletions of fewer than 10 lines of code resulted in defects.

To fully understand these effects of small changes in particular, and changes in general, this study should be replicated across systems in different domains and of different sizes.

7 Discussion of Threats to Validity

There are three types of validity that must be considered in this descriptive and relational study: construct, internal, and external validity.

7.1 Construct Validity

We divide construct validity into three parts [34]: intentional validity, representation validity, and observational validity. In terms of intentional validity, our constructs are well understood and agreed upon in the general context in which this research has been done, and represent what we intend to study.

It is in terms of representational validity that we have a weakness with our change construct in the context of corrective dependent changes. There are three ways a change can be made to fix a fault: Correct the line that was changed, change another line to bring the semantics of the change into full compliance with the originally intended change, or compensate for the fault by fixing another piece of code entirely. For dependent changes we measure only the first of the three representations of change. The remaining two cases would require more information about the relationship among changes than is either available or automatically inferable.

For our definition of dependent changes, our observational validity is very good. Our measure is completely accurate in examining and reporting the version delta data. However, our automated measure for determining the purpose of changes is not as reliable as we would like, but workable in this context.

7.2 Internal Validity

Most of the data we use in this study is explicit in the change and version management systems and can be considered to be completely reliable. The straightforward presentation of the data with a minimum of manipulation strengthens the internal validity for the study.

There is some slight concern about the magnitude of one-line changes because we count only one of the three ways of correcting a fault. However, we argue that while the net affect is that the number of one-line changes automatically determined may be less than the actual number of one-line changes, it does not significantly affect analyses. First, we

believe that the number of compensatory changes is very small and, hence, insignificant in our study. Second, if we assume that one-line changes in error only require one-line changes to fix them, then only 4.5 percent of the one-line changes are not accounted for in our definition of dependent changes (5,701 dependent one-line changes from a total population of 5,969).

Another threat to internal validity in the data we used is the reliability of the automatic classification mechanism. Given that this is always an issue even when using human judgments and that the reliability reported by Mockus and Votta is fairly good, we do not see this as a significant issue.

7.3 External Validity

Meltzoff [33] distinguishes between external validity and generalizability. While external validity and generalizability are often considered to be synonymous, there is merit in considering them to be separate concepts with external validity focused on the consistency of the purpose of, the intent of, or the claims made about a study and the supporting evidence provided by that study. Trying to show that something exists, or to understand and to describe a particular phenomenon (such as the study of small source code changes here) is different from claiming that results generalize to all possible domains of software systems. With respect to this view of external validity, we claim strong external validity—namely, that our results strongly support the purpose and intent of our study. The purpose of this study was to illuminate the rhetoric of small changes in the context of 5ESS and to understand the relationships between the size of, the types of, the purpose of, and the qualitative results of these changes.

It is with respect to generalizability that we cannot make claims as strongly as we would like. The subsystem used for this study is representative of the various subsystems of 5ESS and, thus, can be used a surrogate for the entire system. (cf. [6], [16]). Further, we claim that 5ESS is representative system for large-scale, highly reliable, real-time systems for the following reasons.[6] First, it is built with programming languages (C and C++) and a development environment (UNIX™) commonly used for building and evolving these kinds of systems. Second, these kinds of systems cannot be successfully built without the well-educated and well-trained staff, the well-defined processes, as well as the elaborate management, evolution and development environmental support found in the 5ESS project. A mature organization with mature processes and technology support is a necessary precondition to building and these kinds of systems. One factor about telephony that may not apply in these systems, however, is the amount of new features added with each release. This difference may diminish the generalizability of our results to reliable real-time systems.

The primary problem in generalizability lies in the fact that it is not clear whether these results apply to smaller systems and systems of different domains and applications. Given the size and complexity of the system, we can certainly argue that the problems found here are at least as severe as any found in smaller systems or systems in other domains. The limits of generalizability will only become clear as studies such as this are replicated across a wide variety of domains and systems.

6. Our generalizability claim here is consistent with a variety of studies published on 5ESS— cf. [2], [3], [4], [5], [6], [7], [13], [16], [24].

8 Issues in Replication

In this study, we used data gathered from two repositories: the change management system (ECMS) and the version management system (SCCS). Modern version or configuration management systems such as CVS [28] and ClearCase [29] go beyond the basic functionality of managing version histories and include some of the change management information functions we found in ECMS. The data comparable to what we used that is readily available in these modern version management systems includes:

- the standard version management data
  - version lineage: current version, source version,
  - delta information: lines added, deleted, changed,
  - time of the changes;
- change management information
  - information about the person responsible for the change,
  - text explaining the change and the rationale for it.

The standard version management data enables the size and kinds of changes to lines of code to be calculated and classified. Moreover, the version management data can be used to calculate the dependency data as we have done here. The text explaining the changes and the reasons for those changes can be used to analyze and to classify the purposes of those changes either automatically or manually. One can use the mechanism of Mockus and Votta [3] derived data from and applied to 5ESS (as we did here). Alternatively, one can use the more recent mechanism of Hassan and Holt [30], [31] used on CVS-managed open-source project data. The two mechanisms are comparable with the latter achieving a somewhat higher level of agreement with manual classification than the former.

Without an MR mechanism, it would be difficult to relate logical changes to physical changes as we have done here. Using files as both the logical and physical unit of change instead of MRs would provide congruent data. Thus, exact replication would require the same logical data we used from ECMS. However, there are sufficient data available to perform studies sufficiently congruent to continue exploring the rhetoric of small changes in particular and the phenomena of evolution in general.

9 Recommendations

There are several types of recommendations that stem from this research and may be useful in practice for the building and evolving of software systems.. The first set of recommendations is for version management systems. 1) The notion of a logical set (provided in the case of 5ESS by MRs) of changes is critical for understanding both the rhetoric and the implications of changes. This may be

TABLE 4
χ^2 Analysis of the Purpose of One-Line Changes versus the Entire Set of Changes

		all lines	one-line
corrective	expected	24049	1897
	observed	23687	2259
	chi-square	5.438	68.919
perfective	expected	5801	458
	observed	5764	495
	chi-square	0.24	3.038
adaptive	expected	34017	2684
	observed	34557	2144
	chi-square	8.203	108.599
inspection	expected	6612	522
	observed	6473	661
	chi-square	2.935	37.197
unclassified	expected	1779	140
	observed	1777	142
	chi-square	0.002	0.02

significance	chi-square
p<.2	5.99
p<.1	7.78
p<.05	9.49
p<.025	11.14
p<.01	13.28
p<.001	18.47

total chi-square	df
234.995	4

achieved manually by disciplined and systematic commenting but is easier, systematic, and consistent if wrapped in an interface around the notion of a logical change set on top of your version management system. This is especially important since there are significant numbers of changes made to code that was previously changed. 2) As more change management functions are added to versions management systems, thought should be given to adding several kinds of change classifications. For example, the simple classification as to the purpose of the change is useful and likely to be more reliable than the automated classification techniques used here and elsewhere. This will make it easier to argue for stronger internal validity in future studies as well.

The second set of recommendations is for evolution processes. 1) While it is tempting to take the risk of ignoring small changes as not having significant effects, there is also the fact that small changes (10 lines and under) in this study accounted for approximately 50 percent of the changes to the system. They do, however, have significantly less risk associated with them than larger changes. We recommend a more agile process to assess the small changes and their impact. Indeed, it may be useful to have a changes size sensitive assessment process: one for under 10 lines (46 percent of the files changed and likelihood of generating an error of about 8 percent[7]), one for 20 to 50 lines (20 percent of the files changed with a likelihood of generating an error of about 19 percent) and one for 50 and over (4 percent of the files changed with a likelihood of generating an error of about 35 percent). We base this on the rhetoric shown in the interplay of inserted and modified lines relative to the size of the change. 2) As a matter of practical quality control, particular attention should be paid to modified lines in corrective changes and inserted lines in adaptive changes since they are the most frequently occurring in those two reasons for change, although it is probably easier just to inspect both types of changes in both cases.

7. We arrived at these likelihood estimates by taking the average faults rates for each set of change sizes from the data in Fig. 5.

10 Further Work

Very few studies have been done to understand the software development process by the analysis of changed lines. Most studies have focused on the analysis of changed files or system components at various granularities. While the software project we analyzed had files varying in sizes from 50 lines of code to 50,000 lines of code, we did not consider the individual file sizes separately. Is there a relationship between the size of the file and the probability of error due to change? Our intuition is that changes (irrespective of change size) made to larger files will introduce more errors since the developer is less likely to have a full understanding of these larger files.

In this analysis, we have only considered those defects that were introduced in the lines affected by the change. However, making a change to a part of the code could affect another part of the same file, either very close to the changed lines or in other parts of the program. In the future, we intend to extend this research to study localization effects of making changes.

Finally, to understand fully the small set of changes that result in faults, some of them catastrophic, we need to investigate the context of those changes. Are there common characteristics in the code that is changed? For example, is it in abnormal rather than normal code—studies in interface faults by one of the authors showed that a significant number of faults occurred in error handling code [19], [20]. Are there common characteristics in the changes themselves? Are there domain specific aspects to this set of changes or are they uniform across domains? There is still much work to be done to understand the rhetoric of small changes, indeed of changes of any size.

Appendix

In each of the χ^2 analyses in Tables 4, 5, 6, 7, and 8, there are three matrices. The first is the χ^2 analysis with each cell containing the expected frequency, the observed frequency, and the χ^2 value for that cell (i.e., the value that that cell contributes to the total value for the distribution. The second contains the various χ^2 values for the level of

TABLE 5
χ^2 Analysis of the Types of One-Line Changes versus All Changes

		all lines	one-line
modify	expected	21592	2402
	observed	20286	3708
	chi-square	79.004	710.206
insert	expected	27535	3063
	observed	28974	1624
	chi-square	75.204	676.05
delete	expected	2365	263
	observed	2232	396
	chi-square	7.471	67.163

significance	chi-square
p<.2	3.22
p<.1	4.61
p<.05	5.99
p<.025	7.38
p<.01	9.21
p<.001	13.82

total chi-square	df
1615.098	2

TABLE 6
χ^2 Analysis of Dependent versus Original Change Purpose

		corrective	perfective	adaptive	inspection
inspection	expected	1793	254	4187	621
	observed	1263	117	4584	891
	chi-square	156.568	73.888	37.624	117.28
adaptive	expected	5536	784	12930	1918
	observed	5228	853	13538	1550
	chi-square	17.178	6.007	28.569	70.621
perfective	expected	1153	163	2692	399
	observed	1142	284	2564	418
	chi-square	252.47	89.16	6.128	0.867
corrective	expected	4308	610	10061	1492
	observed	5157	558	9185	1572
	chi-square	167.331	4.486	76.313	4.239

significance	chi-square
p<.2	12.24
p<.1	14.68
p<.05	16.93
p<.025	19.02
p<.01	21.67
p<.001	27.88

total chi-square	df
856.36	9

TABLE 7
χ^2 Analysis of Type versus Purpose of Change Distribution

		corrective	perfective	adaptive	inspection	unclassified
modify	expected	6762	1605	9622	1802	495
	observed	7767	1199	7611	3181	528
	chi-square	149.179	102.649	420.215	1054.68	2.231
insert	expected	9658	2292	13743	2574	707
	observed	9271	3005	14386	1596	716
	chi-square	15.538	221.648	30.128	371.698	0.123
delete	expected	744	177	1059	198	54
	observed	640	731	540	280	41
	chi-square	14.546	1740.753	254.096	33.661	3.317
combination	expected	7122	1690	10134	1898	521
	observed	6609	829	12020	1416	492
	chi-square	36.992	438.898	350.986	122.512	1.627

significance	chi-square
p<.2	15.81
p<.1	18.55
p<.05	21.03
p<.025	23.34
p<.01	26.22
p<.001	32.91

total chi-square	df
5365.577	12

TABLE 8
χ^2 Analysis of Type versus Purpose of Change Distribution of One-Line Changes

		corrective	perfective	adaptive	inspection	unclassified
modify	expected	1469	322	1394	430	92
	observed	1521	304	1264	524	95
	chi-square	1.821	1.001	12.21	20.586	0.076
insert	expected	644	141	611	188	40
	observed	621	71	801	89	42
	chi-square	36.48	34.759	59.267	52.361	0.059
delete	expected	146	32	139	43	9
	observed	117	120	79	48	5
	chi-square	5.837	241.49	25.745	0.636	1.911

significance	chi-square
p<.2	11.03
p<.1	13.36
p<.05	15.51
p<.025	17.53
p<.01	20.09
p<.001	26.12

total chi-square	df
458.544	8

significance (p) defined in the left column. The χ^2 values listed there are useful to understand the significance of the total χ^2 value, as well as for understanding the level of significance of each cell's contribution to the total value. df represents the degrees of freedom and is computed as $(N_1 - 1) \times (N_2 - 1)$.

ACKNOWLEDGMENTS

The authors would like to thank Harvey Siy, Bell Laboratories, Lucent Technologies, for sharing his knowledge of the 5ESS change management process. They also would like to thank Audrus Mockus, Avaya Research Labs, Tom Ball, Microsoft Research, and the reviewers for their contributions and suggestions.

REFERENCES

[1] F. Brooks, *The Mythical Man-Month.* Addison-Wesley, 1975.
[2] M. Leszak, D.E. Perry, and D. Stoll, "Classification and Evaluation of Defects in a Project Retrospective," *J. Systems and Software,* vol. 61, no. 3, pp. 173-187, 2002.
[3] A. Mockus and L.G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," *Proc. Int'l Conf. Software Maintenance,* pp. 120-130, Oct. 2000.
[4] T.L. Graves and A. Mockus, "Inferring Change Effort from Configuration Management Databases," *Proc. Fifth Int'l Symp. Software Metrics,* pp. 267-273, 1998.
[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 Trans. Software Eng.,* vol. 27, no. 1, Jan. 2001.
[6] D.E. Perry and H.P. Siy, "Challenges in Evolving a Large Scale Software Product," *Proc. Int'l Workshop Principles of Software Evolution (ICSE 1998),* Apr. 1998.
[7] A. Mockus and D.M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical J.,* pp. 169-180, Apr.-June 2000.
[8] R. Rogers, "Deterring the High Cost of Software Defects," technical paper, Upspring Software, Inc., Year?
[9] G.M. Weinberg, "Kill That Code!" *Infosystems,* pp. 48-49, Aug. 1983.
[10] D.M. Weiss and V.R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory," *IEEE Trans. Software Eng.,* vol. 11, no. 2, pp. 157-168, Feb. 1985.
[11] M. Lipow, "Prediction of Software Failures," *The J. Systems and Software,* pp. 71-75, 1979.
[12] E.B. Swanson, "The Dimensions of Maintenance," *Proc. Second Int'l Conf. Software Eng.,* pp. 492-497, Oct. 1976.
[13] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.,* vol. 26, no. 7, pp. 653-661, July 2000.
[14] H.E. Dunsmore and J.D. Gannon, "Analysis of the Effects of Programming Factors on Programming Effort," *The J. Systems and Software,* pp. 141-153, 1980.
[15] I.-H. Lin and D.A. Gustafson, "Classifying Software Maintenance," *Proc. IEEE,* pp. 241-247, 1988.
[16] D.E. Perry, H.P. Siy, and L.G. Votta, "Parallel Changes in Large Scale Software Development: An Observational Case Study," *ACM Trans. Software Eng. and Methodology,* vol. 10, no. 3, pp 308-337, July 2001.
[17] L. Hatton, "Reexamining the Fault Density—Component Size Connection," *IEEE Software,* vol. 14, no. 2, pp. 89-97, Mar./Apr. 1997.
[18] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM,* vol. 27, no. 1, pp. 42-52, Jan. 1984.
[19] D.E. Perry and W.M. Evangelist, "An Empirical Study of Software Interface Errors," *Proc. Int'l Symp. New Directions in Computing,* pp. 32-38, Aug. 1985.
[20] D.E. Perry and W.M. Evangelist, "An Empirical Study of Software Interface Faults—An Update," *Proc. 20th Ann. Hawaii Int'l Conf. Systems Sciences,* pp. 113-126, Jan. 1987.
[21] G.J. Holtzmann, "The Logic of Bugs," *Proc. 10th ACM SIGSOFT Symp. Foundations of Software Eng. (FSE-10),* pp. 81-87, Nov. 2002.
[22] K.E. Martersteck and A.E. Spencer, "Introduction to the 5ESS™ Switching System," *AT&T Technical J.,* vol. 64, no. 6, pp. 1305-1314, July-Aug. 1985.
[23] D.C. Carr, A. Dandekar, and D.E. Perry, "Experiments in Process Interface Descriptions, Visualizations and Analyses," *Proc. European Workshop Software Process Technology,* pp. 119-137, 1995.
[24] L.G. Votta and M.L. Zajak, "Design Process Improvement Case Study Using Process Waiver Data," *Proc. European Software Eng. Conf.,* pp. 44-58, 1995.
[25] P.A. Tuscany, "Software Development Environment for Large Switching Projects," *Proc. Sixth Int'l Conf. Software Eng.,* pp. 58-67, Sept. 1992.
[26] M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.,* vol. 1, no. 4, pp. 364-370, Dec. 1975.
[27] M.M. Lehman, D.E. Perry, and J.F. Ramil, "Implications of Evolution Metrics on Software Maintenance," *Proc. Int'l Conf. Software Maintenance,* 1998.
[28] CVS—Concurrent Versions System, http://www.cvshome.org, 2004.
[29] D.B. Leblang, "The CM Challenge: Configuration Management that Works," *Configuration Management: Trends in Software,* W.F. Tichy, ed., John Wiley and Sons, 1994.
[30] A.E. Hassan, "Mining Software Repositories to Assist Developers and Support Managers," PhD thesis, Dept. of Computer Science, Univ. of Waterloo, Feb. 2005.
[31] A.E. Hassan and R.C. Holt, "Studying the Chaos of Code Development," *Proc. WCRE 2003: Working Conf. Reverse Eng.,* Nov. 2003.
[32] S.G. Eick, J.L. Steffen, and E.E. Sumner Jr., "Seesoft—A Tool for Visualizing Line Oriented Software Statistics," *IEEE Trans. Software Eng.,* vol. 18, no. 11, pp. 957-968, Nov. 1992.
[33] J. Meltzoff, *Critical Thinking about Research: Psychology and Related Fields.* Washington D.C.: Psychological Assoc., 1997.
[34] D.E. Perry, "An Empirical Approach to Design Metrics and Judgments," *Proc. New Vision for Software Design and Production Workshop,* Dec. 2001.

Ranjith Purushothaman received the MS degree in electrical and computer engineering from The University of Texas at Austin. He is a lead engineer in the Enterprise Operating Systems group at Dell Inc. His responsibilities include leading the engineering effort to ensure that Windows operating systems are fully compatible with Dell servers. His areas of interest include operating systems, virtualization software, system recovery, and server consolidation.

Dewayne E. Perry is the Motorola Regents Chair of Software Engineering at The University of Texas at Austin (UT Austin) and the director of the Empirical Software Engineering Laboratory (ESEL). The first half of his computing career was spent as a professional programmer and a consulting software architecture and designer. The next 16 years were spent as a software engineering research MTS at Bell Laboratories in Murray Hill, New Jersey. He has been at UT Austin since 2000. His research interests include empirical studies in software engineering, software architecture, and software development processes. He is particularly interested in the process of transforming requirements into architectures and the creation of dynamic, self-managing, and reconfigurable architectures. He is a member of the ACM SIGSOFT and IEEE Computer Society, has been a coeditor-in-chief of Wiley's *Software Process: Improvement and Practice* as well as an associate editor of *IEEE Transactions on Software Engineering*, and has served as organizing chair, program chair, and program committee member on various software engineering conferences.