Towards Understanding Software Evolution: One-Line Changes

Ranjith Purushothaman
Server Operating Systems Group
Dell Computer Corporation
Round Rock, Texas 78682
ranjith_purush@dell.com

Dewayne E. Perry
ECE & UT ARISE
The University of Texas at Austin
Austin, Texas 78712-1084
perry@ece.utexas.edu

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 change impact from the plethora of software artifacts produced during development and maintenance. We present the analysis of the software development process using change and defect history data. Specifically, we address the problem of one-line changes. The studies revealed that (1) there is less than 4 percent probability that a one-line change will introduce an error in the code; (2) nearly 10 percent of all changes made during the maintenance of the software under consideration were one-line change; (3) although the effort for changing one-line of code is smaller compared to larger changes, the vast number of changes result in a significant amount of effort.

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, 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 errors, 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. An area of concern is the issue of software code degrading through time as more and more changes are introduced to it – code decay [5]. While change itself is unavoidable, there are some aspects of change that we can control. One such aspect is the introduction of defects while making changes to software, thus preventing the need for fixing those errors.

A software change has different properties such as size, diffusion, type, duration, etc., and we are interested in studying the impact of the size and type of change on the risk of failure. The initial trigger for our research comes from a claim made by a software guru from industry: *a one-line change has a 50% chance of being wrong* (He did not mean the probability of a one-line change being right or wrong but that half of one-line changes are incorrect). This intuitively seems to be far too high a proportion.

Managing risk is one of the fundamental problems in building and evolving software systems. We deviate from what we know to be the best way to do something in order to reduce costs, effort or elapse times. One such common deviation is not to bother much about one line changes at all. For example, we often skip investigating the implications of one line changes on the system architecture; we do not perform code inspections for one line changes; we may skip unit and integration testing for one line changes; etc. We do this because our intuition tells us that the risk associated with one line changes is small.

However, we all know of cases where one line changes have been disastrous. Gerald 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.

In either case, innocuous or disastrous, we have very little actual data on the one line changes and their effects to support our decisions. We base our decisions about risk on intuition and anecdotal evidence at best.

Our approach is different from most other studies that address the issue of software errors because we have based the analysis on the property of the change itself rather than the properties of the code that is being changed [7]. Change to software can be made by addition of new lines, modifying existing lines, or by deleting lines. We expect each of these different types of change to have different risks of failure.

Our main hypothesis is that the probability of a one-line change resulting in an error is small. Our second hypothesis is that the failure probability is higher when the change involves adding new lines than either deleting or modifying existing lines of code.

To test our hypotheses, we used data from the source code control system (SCCS) of a large scale software project (5ESS). The Lucent Technologies 5ESS™ switching system software is a multi-million line distributed, high availability, real-time telephone switching system software that was developed over two decades [6]. The source code of the 5ESS project, mostly written in the C programming language, underwent several hundred thousand changes.

The use of data from a generic version control system for our analysis ensures that our results can be extended to any commercial software product. 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 changes to software has not been done before.

In the next section we provide an insight into the past research that has addressed issues related to our analysis. In section 3, we provide the background for the study, describing the change data and the methodology employed for our research. In section 4, we describe our approach for the analysis of the changed lines, focusing first on how we prepared the data. In section 5 we discuss the results of our analysis, and finally conclude the paper in section 6.

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 design and development phases. Quite the contrarily, statistical data shows that maintaining two to ten year old software systems demand possibly as high as 40 percent to 70 percent of the total development effort [15]. We suspect that the number is actually much higher than that. Software maintenance still remains as a difficult process to understand and manage.

Understanding the need for classification of the software changes, E. B. Swanson [12] proposed that change be classified to belong 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 this translates into new development and new features. Perfective maintenance is applied to eliminate inefficiencies, enhance performance, 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.

In his analysis, Les Hatton [17] relates the defect frequency to file size. He states 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] state that, in general, process measures based on change history are more useful in predicting fault rates than product metrics of the code. They give 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] have studied the relation between the size of the change and probability of error and have 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 conclude that the probability of error is much more 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 $\alpha = .05$) that shows a direct relationship between the amount of program changes and the error occurrences.

In the analysis done by Stoll 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 that would involve modification 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 check 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 observational and relational study of one line 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 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 of the type of change on the software evolution process.

The 5ESS™ change history data 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], to identify the reasons for software changes [3], for 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 analysis based on this data.

3. Background – Change Data Description

Traditionally, analysis of software development processes use specific experiments and instrumentation that can limit the scope of the results of the analysis. Hence, to ensure that the results of this analysis are not constrained to just the system under study, data from a well known version control system has been used for this research. Our experimental design could be easily replicated across a wide range of system domains and applications.

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. Change Process

In the 5ESS change management process, a logical change to the system is implemented as an initial modification request (IMR) by the IMR Tracking System (IMRTS). The change history of the files is maintained using the Extended Change Management System (ECMS) for initiating and tracking changes and the Sources Code Control System for managing different versions of the files. Hence, to keep it manageable, each IMR is organized into a set of maintenance requests (MR) by the ECMS as shown in Figure.1 [3][5][7]. The ECMS records information about each MR. Each MR is owned by a developer, who makes changes to the necessary files to implement the MR. Every change that is made is recorded by the SCCS in the form of a single *delta*. Each delta provides information on the following attributes of the change: 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. Hence there are typically many more records in the delta relation than there are files that have been modified by an MR.

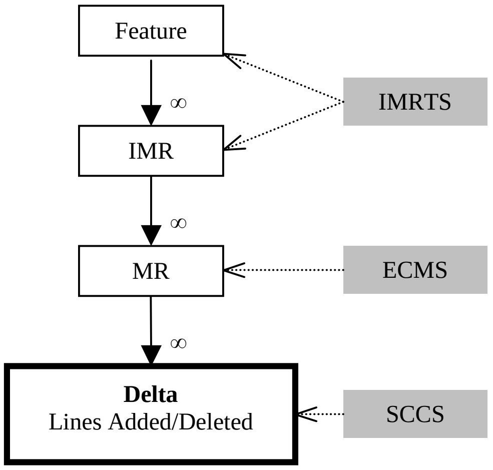

Figure 1: **Change hierarchy**

3.2. Change Data

The 5ESS™ source code is organized into subsystems, and each subsystem is subdivided into a set of modules. Any given module contains a number of source lines of code. For this research, we use data from one of the subsystems of the project. The Office Automation (OA) subsystem contains 4550 modules that have a total of nearly 2 million lines of code. Over the last decade, the OA subsystem had 31884 modification requests (MR) that changed nearly 4293 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 the 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.
- A modification to a single/multiple lines accompanied by insertion or/and deletion of one or more lines.

The following changes would qualify to be a one-line change:

- One or more modifications to a single line.
- One or more lines replaced by a single line.
- One new statement inserted between existing lines.
- 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. While it was possible to make all changes that are required to be made for a MR in a file in a single delta, developers often performed multiple deltas on a single file for an MR. Hence there were lot more delta records than the number of files that needed to be modified by MRs.

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. For example, in Table 1, the MR *oa101472pQ* 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 3 lines of the file. With nearly 32000 MRs that modified nearly 4300 files in the OA subsystem, the aggregation of the changes made to each file at the MR level gave us 72258 change records for analysis.

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

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 done by Mockus and Votta [3]. 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.

However, there were numerous instances when changes made could not be classified clearly. For example, certain changes were classified as 'ICC' 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'

Table 2: Change Classification (purpose)

ID	Change type	Change purpose
B	Corrective	Fix defects

C	Perfective	Enhance performance
N	Adaptive	New development
I	Inspection	Following inspection

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 perfective 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, adaptive or perfective changes.

Table 3: Change classification (implementation)

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

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 (insert and delete). However, for every changed file SCCS maintains an SCCS file that relates the MR to the insertions and deletions made to the actual module. Scripts were used to parse these 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.

4.3 Identifying file 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. 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*.

The dependency, as we have defined earlier, may have existed due to bug fixes (corrective), enhancements (perfective), changes from inspection, or new development (adaptive). 2530 files in the OA subsystem were found to have undergone dependent change. 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 51478 dependent change records and this data was the core of our analysis.

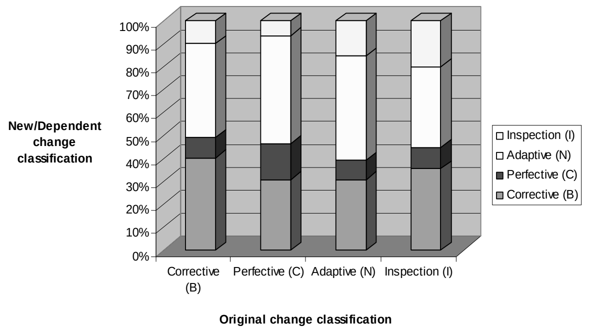

Figure 2: **Distribution of change classification on dependent files**

In Figure 2, 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 implementation type. From the distribution it can be noted that most bug fixes were made to code that was already changed by an earlier MR to fix bugs. At this point of time, we can conclude that *roughly 40 percent of all changes made to fix bugs introduced more bugs*.

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.

5. Results and Analysis

The analysis of the data proceeds in several steps. We begin with an investigation of the software project based on the change size.

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. Figure 3 shows the distribution of the changed files based on the number of lines that were changed. The vertical axis shows the percentage of changed files that changed the number of lines specified on the horizontal axis.

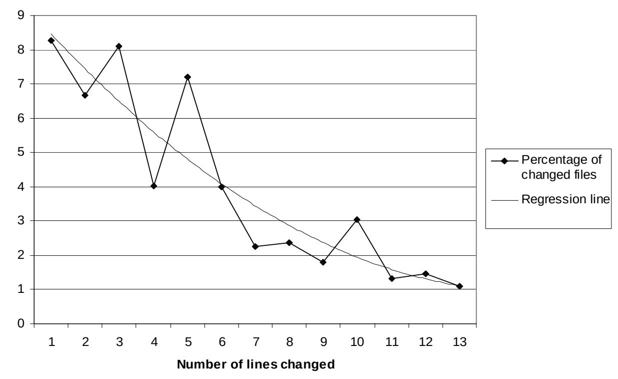

Figure 3: **Distribution of small changes**

From Figure 3, we can see that *nearly 10 percent of changes 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 changes involved changing less than 10 lines of code*.

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, Figure 4 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% of all changes were those that changed less than 50 lines of code.*

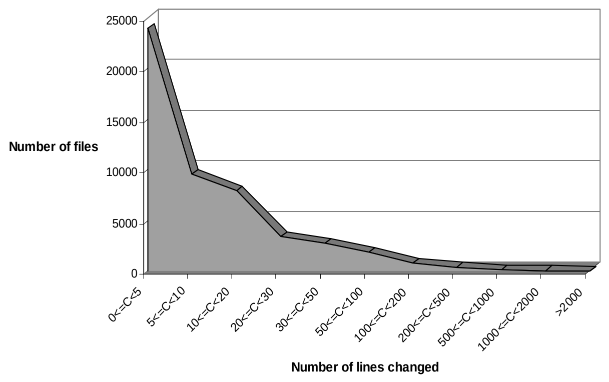

Figure 4: **Change size distribution across files**

5.2. Erroneous changes

We next analyze those changes that resulted in errors. In Figure 5, 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 error.*

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, it provides more avenues for the developer are provided to make mistakes. These increased opportunities to introduce errors are likely due to an increase in the number of possible interactions.

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. Figure 6 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.*

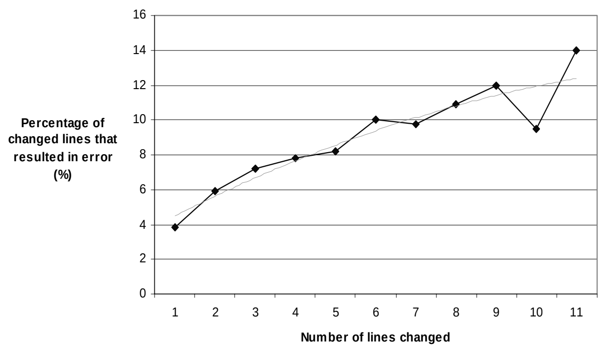

Figure 5: **Errors introduced by change**

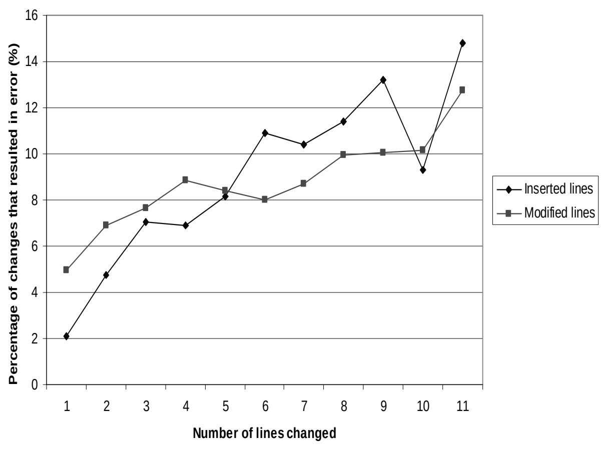

Figure 6: **Erroneous changes classified by type of change**

From Figure 6, 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 5 lines are changed; inserted new lines introduce more errors with larger change sizes.

To emphasize this behavior, in Figure 7, 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 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.

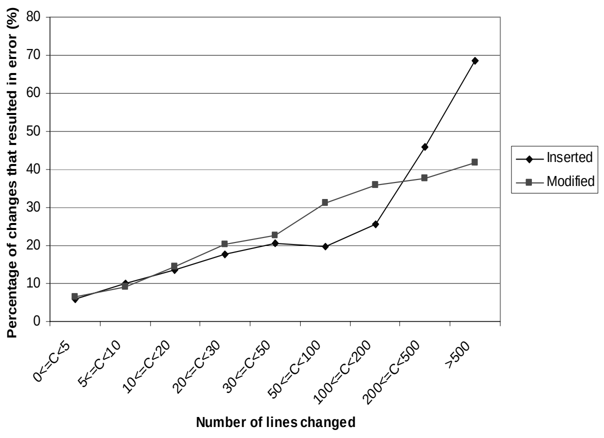

Figure 7: **Erroneous changes versus change size**

5.3. Change Process Metrics

How are the types of change related to change classifications? In Figure 8, 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.

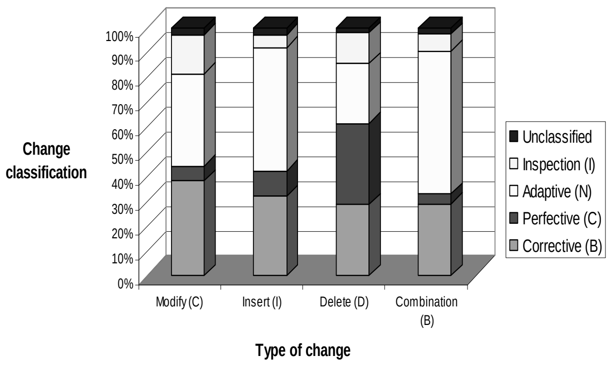

Figure 8: **Relation between change classification and change type**

We can see that the Figure 8 holds no surprises except maybe that deletion of lines occurred pretty much uniformly for adaptive, corrective and perfective changes. Note, however, that there are more deleted lines than modified, inserted and combined in perfective evolution.

Figure 9 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 Figure 8 and Figure 9 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.

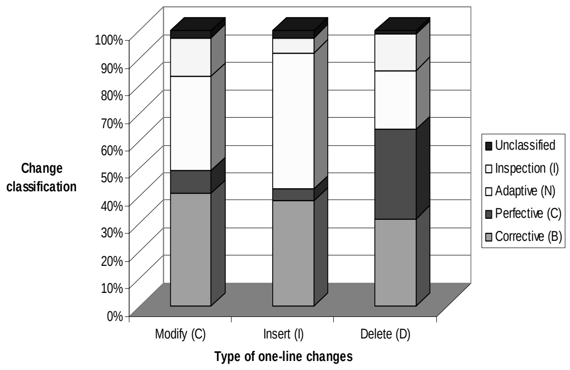

Figure 9: **Relation between various change types for one-line changes**

In the figures 10 and 11, we show the distribution of the OA subsystem change data across the different change classifications that were defined earlier. We can see that the *maximum number of changes was made for adaptive purposes and most changes were made by inserting new lines of code.*

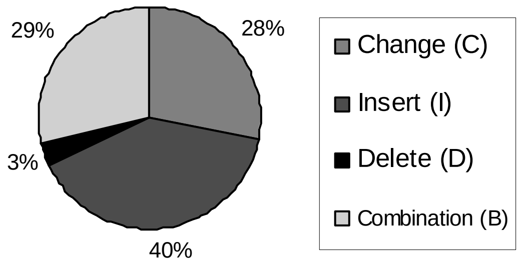

Figure 10: **Distribution of changes based on type**

5.4. Validity and Replicability

There are three types of validity that must be considered in this observational and relational study: construct, internal, and external validity. Our constructs are well understood and agreed upon in the general context in which this research has been done. Furthermore, the observable measures presented here represent the intended constructs.

The straightforward presentation of the data with a minimum of manipulation supports our claim for good internal validity for the study.

It is in the case of external validity 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]). The weakness in our claim for external validity lies in the fact that while it is a representative system for large, real-time systems and is built with a commonly used programming language and development environment, it is not clear how well it represents 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. Thus while it is not as generalizable as we might like, it is an important fist step in understanding one line changes and makes a significant contribution to our understanding of evolution.

Given that modern SCM systems now include change management facilities in addition to the historical version management facilities, we argue that our study should be easily replicable using systems of differing sizes and domains.

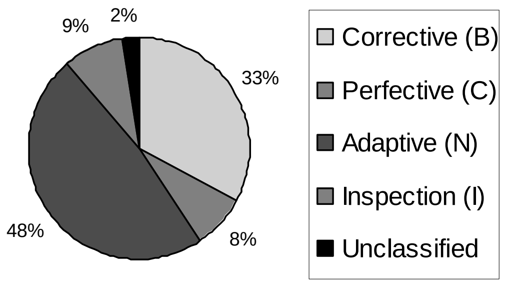

Figure 11: **Distribution of changes based on purpose**

6. Conclusions and Next Steps

We have found that the probability that a one-line change would introduce at least 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 that the intial claim: "*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 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 of time after the initial release of the software product.
- Nearly 40 percent of the changes that were made to fix bugs introduced one or more other bugs in the software.
- Roughly 50 percent of the changes involve changing less than 10 lines of code. 95 percent of changes change less than 50 lines of code.
- Nearly 10% of all the changes made were one line changes.

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

7. Future Work

Very few studies have been done to understand the software development process by the analysis of changed lines. While the software project we analyzed had modules varying in sizes from 50 lines of code to 50,000 lines of code, we did not consider the individual module sizes separately. Is there a relationship between the size of the module 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 an understanding of the larger modules.

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 module, 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? Etc.

8. Acknowledgements

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

9. References

[1] Fred Brooks, "The Mythical Man-Month", Addison-Wesley, 1975

[2] Dieter Stoll, Marek Leszak, Thomas Heck, "Measuring Process and Product Characteristics of Software Components – a Case study"

[3] Audris Mockus, Lawrence G. Votta, "Identifying Reasons for Software Changes using Historic Databases", In International Conference on Software Maintenance, San Jose, California, October 14, 2000, Pages 120-130

[4] Todd L Graves, Audris Mockus, "Inferring Change Effort from Configuration Management Databases", Proceedings of the Fifth International Symposium on Software Metrics, IEEE, 1998, Pages 267-273

[5] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, Audris Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data", IEEE Transactions on Software Engineering, Vol. 27, No. 1, January 2001

[6] Dewayne E. Perry, Harvey P. Siy, "Challenges in Evolving a Large Scale Software Product", Proceedings of the International Workshop on Principles of Software Evolution, 1998 International Software Engineering Conference, Kyoto, Japan, April 1998

[7] Audris Mockus, David M. Weiss, "Predicting Risk of Software Changes", Bell Labs Technical Journal, April-June 2000, Pages 169-180

[8] Rodney Rogers, "Deterring the High Cost of Software Defects", Technical paper, Upspring Software, Inc.

[9] G. M. Weinberg, “Kill That Code!”, Infosystems, August 1983, Pages 48-49

[10] David M. Weiss, Victor R. Basili, “Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory”, IEEE Transactions on Software Engineering, Vol. SE-11, No. 2, February 1985, Pages 157-168

[11] Myron Lipow, “Prediction of Software Failures”, The Journal of Systems and Software, 1979, Pages 71-75

[12] Swanson. E. B., “The Dimensions of Maintenance”, Procedures of the Second International Conference on Software Engineering, San Francisco, California, October 1976, Pages 492-497

[13] Todd L. Graves, Alan F. Karr, J.S. Marron, Harvey Siy, “Predicting Fault Incidence Using Software Change History”, IEEE Transactions on Software Engineering, Vol. 26, No. 7, July 2000, Pg 653-661

[14] H.E. Dunsmore, J.D. Gannon, “Analysis of the Effects of Programming Factors on Programming Effort”, The Journal of Systems and Software, 1980, Pages 141-153

[15] Ie-Hong Lin, David A. Gustafson, “Classifying Software Maintenance”, 1988 IEEE, Pages 241-247

[16] Dewayne E. Perry, Harvey P. Siy, Lawrence G. Votta, “Parallel Changes in Large Scale Software Development: An Observational Case Study”, ACM Transactions on Software Engineering and Methodology 10:3 (July 2001), pp 308-337.

[17] Les Hatton, Programming Research Ltd, “Reexamining the Fault Density – Component Size Connection”, IEEE Software, March/April 1997, Vol. 14, No. 2, Pages 89-97

[18] Victor R. Basili, Barry T. Perricone, “Software Errors and Complexity: An Empirical Investigation”, Communications of the ACM, January 1984, Vol 27, Number 1, Pages 42-52

[19] Dewayne E. Perry and W. Michael Evangelist. ``An Empirical Study of Software Interface Errors'', *Proceedings of the International Symposium on New Directions in Computing*, IEEE Computer Society, August 1985, Trondheim, Norway, pages 32-38

[20] Dewayne E. Perry and W. Michael Evangelist. ``An Empirical Study of Software Interface Faults --- An Update'', *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, January 1987, Volume II, pages 113-126.