Towards Understanding the Rhetoric of Small Changes
-- Extended Abstract --

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

Dewayne E. Perry
Electrical & Computer Engineering
The University of Texas at Austin
Austin, Texas 78712
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 small 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 changes; (3 the phenomena of change differs for additions, deletions and modifications as well as for the number of lines affected.

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.

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. 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 small 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 first hypothesis is specific to one-line changes, namely 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. The Lucent Technologies 5ESS™ switching system software is a multi-million line distributed, high availability, real-time telephone switching system that was developed over two decades [6]. The source code of the 5ESS project, mostly written in the C programming language, has undergone several hundred thousand changes.

Our primary contribution in this empirical research is an initial descriptive and relational study of small changes. 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.

2. Background – Change Data Description

In the 5ESS, a *feature* is the fundamental unit of system functionality. Each feature is implemented by a set of Initial Modification Requests (IMRs) where each IMR represents a logical problem to be solved. Each IMR is implemented by a set of Modification Requests (MRs) where each MR represents a logical part of an IMR's solution. The change history of the files is maintained using the Extended Change Management System (ECMS) (as shown in Figure.1 [3][5][7]) for initiating and tracking changes and the Sources Code Control System for managing different versions of the files. 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. The changes themselves are maintained by SCCS in the form of one or more *deltas* depending on the way the changes are committed by the developer. Each delta provides information on the 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.

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; and,
- 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 when an MR consists of either:

- One or more modifications to a single line;
- One or more lines replaced by a single line;
- 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.

3. 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.

3.1 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

3.2. 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 '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'

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 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, 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 (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 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.

3.3 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.

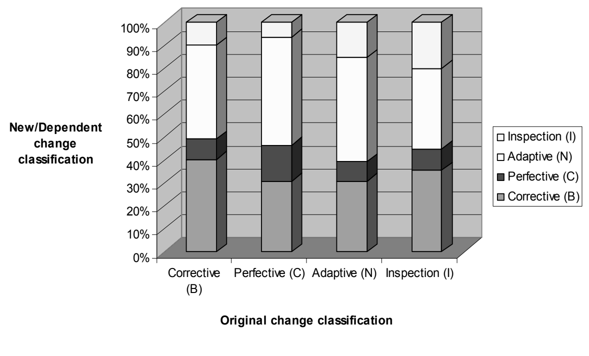

Figure 1: **Distribution of change classification on dependent files**

In Figure 1, 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 errors introduced more errors*.

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.

4. Analysis Summary 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 the intial claim: "*one-line changes are erroneous 50 percent of the time*" [21]. 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.

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 in time after the initial release of the software product.
- Nearly 40 percent of the 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 fewer than 10 lines of code; nearly 95% 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 towards 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.

5. 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.

6. 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.

[21] Anecdotally related in an email conversation.