# Taking Lessons from History

**— Research Abstract —**

Thomas Zimmermann
Department of Computer Science
Saarland University, Saarbrücken, Germany
tz@acm.org

## ABSTRACT

Mining of software repositories has become an active research area. However, most past research considered any change to software as beneficial. This thesis will show how we can benefit from a classification into good and bad changes. The knowledge of bad changes will improve defect prediction and localization. Furthermore, we will describe how to learn project-specific error patterns that will help reducing future errors.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*corrections, version control, reverse engineering*; D.2.8 [**Metrics**]: Complexity measures, Process metrics, Product metrics

## General Terms

Management, Measurement, Reliability

## 1. INTRODUCTION

*The only real mistake is the one from which we learn nothing.*
*—John Powell*

Nowadays, software development produces a huge amount of information: changes to source code are recorded in version archives, bugs are reported to problem databases, and development is discussed in mailing lists and newsgroups. Recently, a new research area called *mining software repositories* has emerged. It showed that historical data is a valuable asset when it comes to understanding change tasks [4], guiding programmers [23, 19], and identifying logical coupling [7] of huge software systems.

The common theme of research in this area are *changes* to source code. A change can be caused by anything: a new feature, refactoring, or a bug fix. Furthermore any change has impact on a system: it can introduce or correct defects[1] or it can cause test cases to fail or pass. However, most existing research did not take this *quality* of changes into account: all changes were considered *good*.

[1]We use the term *defect* to refer to an error in the source code, and the term *failure* to refer to an observable error in the program behavior.


*ICSE'06*, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

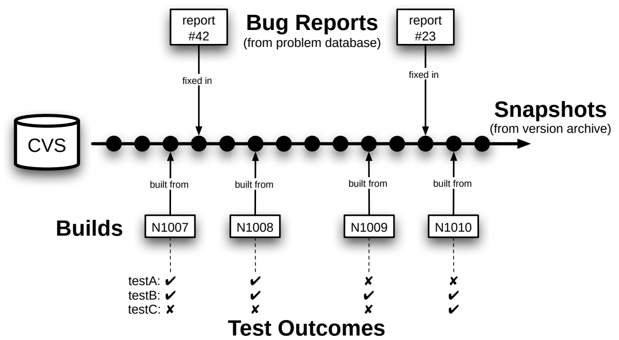


**Figure 1: Data that is available for projects.**

This thesis will leverage the quality aspect of changes. It will first develop ways to assess changes with respect to defects and test cases. Next it will show the benefits of this classification into *good* and *bad* changes. For instance, the knowledge of bad changes will improve defect prediction and localization. Furthermore, it will describe how to learn project-specific error patterns that will help reducing future errors.

The underlying research hypothesis is as follows: *When mining software repositories, we can leverage the knowledge of (past) bad changes for defect localization, defect prediction, and for finding error patterns—in short, "learning from mistakes".*

## 2. LEARNING FROM MISTAKES

Figure 1 shows the data sources that are available for most software projects. We will use version archives to build *snapshots* and to record changes between these snapshots. Using the textual description of changes, we automatically assign *bug reports* to snapshots. Additionally, we collect *builds* for the snapshots. Builds will be used for dynamic program analysis and the execution of *test cases*.

The proposed strategy for mining this data consists of three steps:

1. *Record changes.* Before we can learn from changes we need some way to represent them. We will use *tokens* to describe what has been changed within an element (Section 2.1).
2. *Classification of changes.* Additionally, we distinguish between *good* and *bad* changes (Section 2.2).
3. *Learn from changes.* We can use changes and their classification to predict future failures and to characterize and locate defects (Section 2.3).

Finally, we will evaluate all techniques developed in this thesis with the data available from open source projects (Section 2.4).

| Token type | For what? | What is captured? |
|---|---|---|
| Modifier | modifier | public, private, final, ... |
| Call | method call | method name and signature |
| Name | variable usage | variable name |
| Type | variable usage | variable type |
| Throws | method declaration | thrown exception |
| Throw | throw statement | thrown exception |
| Catch | catch expression | caught exception |
| Keyword | keywords | if, for, while, ... |
| Extends | type declaration | extended type |
| Implements | type declaration | implemented interface |
| Import | import statement | imported class/package |

**Table 1: Different kinds of tokens**

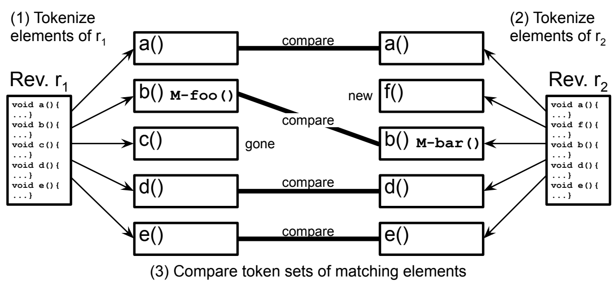


**Figure 2: Comparing two revisions $r_1$ and $r_2$ of a file**

## 2.1 Recording Changes

Previous research focused on the *location* of a change—such as files [2], classes [3, 7], or methods [21]—and on *properties* of changes—such as number of lines changed, developers, or whether a change is a fix [12].

In this thesis, we will additionally investigate changes at the level of *tokens*. A token represents some syntactic content of an element. As Table 1 shows, we distinguish between different kinds of tokens: For methods, we capture method calls, variable usages, and exception handling; for classes, we capture inheritance relations; for compilation units, we capture imported classes.

Using tokens, it is straightforward to compute fine-grained changes between two revisions $r_1$ and $r_2$ (see Figure 2). First, we represent each element of revision $r_1$ as a multiset of tokens; we do the same for the elements of revision $r_2$. Finally, we compare the multisets of matchings elements. As a result we get differences such as in method b() one call to method foo() was deleted and one call to method bar() was inserted. Other possible changes that we can detect are "two usages of String variables were deleted" and "one throw statement for EmptyStackException was added".

The usage of tokens is motivated by the research of Li and Zhou who inferred implicit programming rules based on method call and variable type tokens. They identified several violations of these rules which turned out to be defects [9].

## 2.2 Classification of Changes

In addition tokens that represent changes, this thesis will leverage the *nature* of a change, i.e., the reason of a change and its impact on the software system. There are several, orthogonal ways to classify changes:

**Adaptive, corrective, and perfective changes.** Mockus and Votta proposed a classification into three categories [11]:

- changes that add new features (*adaptive*),
- changes that correct defects (*corrective*), and
- changes that restructure code to accommodate future changes which also includes refactoring (*perfective*).

Mockus and Votta also presented an algorithm to categorize changes based on their textual description. However, since the quality of these descriptions differs widely among projects, their algorithm is not always applicable.

Recent research concentrated on inferring links to problem databases such as BUGZILLA [4, 6] to gather additional information about corrective changes.

**Fix-inducing changes.** In our previous work we defined the concept of *fix-inducing* changes [18]. A change $\delta_b$ induces a fix $\delta_f$, if one of the lines introduced by $\delta_b$ is corrected later on by $\delta_f$. Fix-inducing changes *need not* to correspond with the introduction of a defect, rather, they should be understood as an indicator for the stability of a change.

**Test fail/pass-inducing changes.** Many software projects use regression tests in their build process. We classify changes with respect to their impact on such tests:

- Changes that flip a test case from pass (✔) to fail (✘) are called *test fail-inducing*. In Figure 1 such a change occurs between builds N1008 and N1009 for test testA.
- Changes that flip a test case from fail (✘) to pass (✔) are called *test pass-inducing*. In Figure 1 such a change occurs between builds N1009 and N1010 for test testC.

A change can be both test fail- and pass-inducing at the same time, but only for different test cases.

In practice most projects perform regression tests on a daily or weekly basis. Since many changes occur between the individual test runs, we need an additional analysis such as *change impact analysis* [16] or *delta debugging* [20] to identify those changes that actually affected the outcome of a test case. In the presence of *continuous testing* [17] we can do without such an analysis and identify test fail/pass-inducing changes directly from test outcomes.

## 2.3 Possible Applications

There are several possible applications that use tokenized changes and their classification.

**Prediction.** When a change is fix-inducing, this indicates that is has been unstable. Therefore, we used the percentage of fix-inducing changes for a location to define the *risk of change*. The higher this risk for a location, the more likely a change in that location has to be fixed later on. Future risk of change can be either predicted from past risk, metrics, or a combination of source code tokens [22].

*Post-release failures* are failures that are observed within the first six months after a release. Such failures are of particular interest for any commercial software project because they may harm the consumers' trust in a product. This thesis will investigate whether tokenized changes predict post-release failures.

| Project | LOC | Developers |
|---|---|---|
| ARGOUML (UML editor) | 128,915 | 24 |
| ASPECTJ (compiler) | 558,145 | 12 |
| AZUREUS (file-sharing client) | 242,614 | 14 |
| COLUMBA (mail client) | 103,424 | 26 |
| ECLIPSE (development environment) | 1,766,528 | 139 |
| JEDIT (editor) | 562,984 | 122 |

**Table 2: Projects that will be used for the evaluation. LOC were generated using David A. Wheeler's "SLOCCount".**

**Characterization of Changes.** This thesis will explore whether we can leverage tokenized changes for *change classification*. We expect to improve on the classical change classification algorithm proposed by Mockus and Votta [11], since our token-based approach looks on the change itself rather than on its textual description.

Using machine learning techniques such as frequent pattern mining [1], we will identify *patterns* that are characteristic for the classes of changes defined in Section 2.2. For instance, we will learn project-specific *corrective* (or error) patterns that help us to understand errors that are common within a particular project.

**Defect Localization.** Method calls that are added simultaneously to source code form a pattern. We leveraged this observation to mine project-specific usage patterns and searched for their violations dynamically [10]. With several new kinds of tokens and several classes of changes, we expect to get different and more specific patterns that locate defects more precisely.

Furthermore, we will train classifiers for test fail/pass-inducing changes and apply them to changes that are yet untested. We are confident that this will help to identify changes that are likely to cause failures.

## 2.4 Planned Evaluation

We will use *open source projects* for our evaluation, such as the ones listed in Table 2. These projects provide all necessary data, such as version archives, problem databases, test suites, and builds. Furthermore, they are of considerable size and developed by many programmers. For the evaluation we will split the project histories into a *training* and a *testing* phase and use standard measures, such as precision, recall, and correlation, to assess the accuracy of our approach.

For the prediction part, we will additionally use the *NASA Metrics Data Program* [13] to cross-check our results where possible. The NASA Metrics Data Program contains software metrics and associated defect data at the method level for closed source projects.

# 3. EXPECTED CONTRIBUTIONS

The contributions of this thesis will likely be the following:

**A classification into good and bad changes.**

*"The addition of a call to the method handler() in line 42 caused a test case to fail and is bad."*

This classication will be with respect to defects and test cases and improve on existing ones [11] by focusing on the impact of a change rather than on its purpose.

**A technique to mine project-specific error patterns.**

*"Constructing a BankTransaction object and calling begin() on this object without calling commit() leads to errors."*

This technique will be more general than existing static analysis approaches and statistical techniques [5] as it is not a-priori limited to a particular set of pattern templates. Furthermore, it will focus on project-specific rather than on application-specific error patterns. Previous research addressed errors in J2EE applications [15] and operating system code [8].

**An improved prediction and localization of defects.**

*"The package com.foo.bar will cause most of the program's post-release failures."*

Although software repository information has been used for defect prediction [14], no one leveraged the notion of bad changes so far. Our research will focus on how well the knowledge of bad changes enhances or performs against competing techniques.

*"The class Broker contains a defect because a call to method commit() is missing."*

Additionally, we can use error patterns or classifiers to locate existing defects and to warn developers when they are likely introducing a new defect.

# 4. ACKNOWLEDGMENTS

My research is funded by the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme". I would like to thank my advisor Andreas Zeller, my PhD soulmate Stefanie Scherzinger, and the reviewers of the ICSE Doctoral Symposium committee for their comments on this research abstract.

# 5. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Data Bases Conference (VLDB)*, pages 487–499. Morgan Kaufmann, 1994.

[2] J. Bevan and J. Whitehead. Identification of software instabilities. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 134–143, Victoria, British Columbia, Canada, Nov. 2003. IEEE.

[3] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. In *Proc. 11th International Workshop on Program Comprehension*, pages 44–53, Portland, Oregon, May 2003.

[4] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, May 2003.

[5] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[7] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Helsinki, Finland, Sept. 2003.

[8] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 69–82, Berlin, Germany, 2002.

[9] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of European Software Engineering Conference/ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 306–315, New York, NY, USA, 2005. ACM Press.

[10] V. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proc. Joint European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 296–305, Lisbon, Portugal, Sept. 2005.

[11] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 120–130, San Jose, California, USA, Oct. 2000. IEEE.

[12] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April–June 2000.

[13] NASA. Metrics Data Program. `http://mdp.ivv.nasa.gov/index.html`.

[14] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[15] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved. SABER: Smart Analysis Based Error Reduction. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 243–251, Boston, MA, July 2004.

[16] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA*, pages 432–448. ACM, 2004.

[17] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 281–292. IEEE Computer Society, 2003.

[18] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? On Fridays. In *Proc. International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, USA, May 2005.

[19] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, Sept. 2004.

[20] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proc. of Joint European Software Engineering Conference (ESEC) and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 253–267. Springer Verlag, 1999.

[21] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 73–83, Helsinki, Finland, Sept. 2003.

[22] T. Zimmermann, J. Śliwerski, and A. Zeller. Locating the risk of change. Technical report, Saarland University, 2006.

[23] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.