Changes and Bugs – Mining and Predicting Development Activities

Thomas Zimmermann
Microsoft Research
Redmond, WA, USA
tzimmer@microsoft.com

Abstract

Software development results in a huge amount of data: changes to source code are recorded in version archives, bugs are reported to issue tracking systems, and communications are archived in e-mails and newsgroups. We present techniques for mining version archives and bug databases to understand and support software development.

First, we introduce the concept of co-addition of method calls, which we use to identify patterns that describe how methods should be called. We use dynamic analysis to validate these patterns and identify violations. The co-addition of method calls can also detect cross-cutting changes, which are an indicator for concerns that could have been realized as aspects in aspect-oriented programming.

Second, we present techniques to build models that can successfully predict the most defect-prone parts of large-scale industrial software, in our experiments Windows Server 2003. This helps managers to allocate resources for quality assurance to those parts of a system that are expected to have most defects. The proposed measures on dependency graphs outperformed traditional complexity metrics. In addition, we found empirical evidence for a domino effect, i.e., depending on defect-prone binaries increases the chances of having defects.

1. Introduction

The amount of data generated during software development is continuously increasing. According to the web-site CIA.vc every 26 seconds a change is reported for an open-source project. As of February 2008, the software development community SourceForge.net hosted 169,383 projects. Besides change, another constant in software development is to err. The bug databases of ECLIPSE and MOZILLA combined contain more 600,000 issue reports.

The availability of all this data recently led to a new research area called *mining software repositories (MSR)*. Both software practitioners and researchers alike use such data to understand and support software development and empirically validate novel ideas and techniques. A detailed survey on mining software repositories techniques was conducted by Kagdi et al. [5]. As they show, research on MSR is very inter-disciplinary. Commonly used techniques come from applied statistics, information retrieval, artificial intelligence, social sciences, and software engineering. Their purpose is very diversified, ranging from empirical studies and change prediction to the development of tools in order to support programmers.

Two examples for MSR applications are *project memories* and *recommender systems.*

Project memories. The HIPIKAT tool recommends relevant software development artifacts, such as source code, documentation, bug reports, e-mails, changes, and articles based on the context in which a developer requests help. The project memory is built automatically and is useful in particular for newcomers [3]. The BRIDGE project at Microsoft is a comparable project within an industrial setting [11].

Recommender systems. Just like Amazon.com suggests related products after a purchase, the EROSE plug-in for Eclipse guides programmers based on the change history of a project. Suppose a developer changed an array `fKeys[]`. EROSE then suggests to change the `initDefaults()` function—because in the past, both items always have been changed together. If the programmer misses to commit a related change, EROSE issues a warning [14]. While EROSE operates on change history as recorded in CVS, more recent tools relied on navigation data [4, 10].

This dissertation makes two contributions to the body of MSR research (see overview in Figure 1). First, it mines fine-grained change for usage patterns and cross-cutting concerns (Section 2). Next, it shows how to predict defects from dependency data, which helps managers to allocate quality assurance resources to the parts of a software that need it most (Sections 3 and 4).

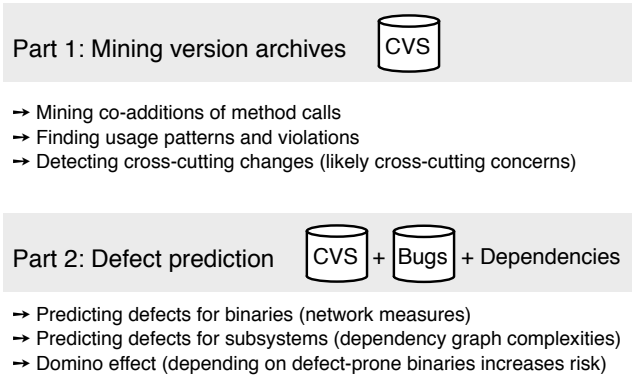

Figure 1. This thesis mines version archives and predicts software defects.

2. Co-Addition of Method Calls

The first part of this dissertation mines version archives for fine-grained changes, more precisely for *co-addition* of method calls, which is when two or more invocations to methods are introduced in the same CVS transaction.

Mining usage patterns. A great deal of attention has always been given to addressing software bugs such as errors in operating system drivers or security bugs. However, there are many other lesser known errors specific to individual applications or APIs and these violations of application-specific coding rules are responsible for a multitude of errors.

We propose DYNAMINE (see Figure 2), which is a tool that analyzes version archives to find highly correlated method calls (usage pattern). Potential patterns are passed to a dynamic analysis tool for validation. The combination of mining software repositories and dynamic analysis techniques proves effective for discovering new application-specific patterns and for finding violations in very large applications with many person-years of development [7].

Mining cross-cutting concerns. Aspect mining identifies cross-cutting concerns in a program to help migrating it to an aspect-oriented design. Such concerns may not exist from the beginning, but emerge over time. By analyzing where developers add code to a program, our history-based aspect mining (HAM) identifies and ranks cross-cutting concerns. HAM scales up to industrial-sized projects: for example, we were able to identify a locking concern that cross-cuts 1,284 methods in ECLIPSE. Additionally, the hit rate of HAM is high; for ECLIPSE, it reaches 90% for the top-10 candidates [1].

3. Defect Prediction

The second part of this dissertation additionally takes information from bug databases into account and moves to an industrial setting. In software development, resources for quality assurance are limited by time and by cost. In order to allocate resources effectively, managers need to rely on their experience backed by complexity metrics [2]. However, often dependencies exist between various pieces of code over which managers may have little knowledge. These dependencies can be constructed as a low level graph of the entire system.

Predicting defects for binaries. We propose to use network analysis on dependency graphs to predict the number of defects for binaries. In our evaluation on Windows Server 2003, we found the recall for models built from network measures is by 10% points higher than for models built from complexity metrics. In addition, network measures could identify 60% of the binaries that the Windows developers considered as critical—twice as many as identified by complexity metrics [13].

Predicting defects for subsystems. We investigated the architecture and dependencies of Windows Server 2003 to show how to use the complexity of a subsystem's dependency graph to predict the number of failures at statistically significant levels [12].

Our techniques allows managers to identify central program units that are more likely to face defects. Such predictions can help to allocate software quality resources to the parts of a product that need it most, and as early as possible.

4. Domino Effect

In 1975, Randell defined the domino effect principle [9]:

> *"Given an arbitrary set of interacting processes, each with its own private recovery structure, a single error on the part of just one process could cause all the processes to use up many or even all of their recovery points, through a sort of uncontrolled domino effect."*

Restating Randell on dependency relationships, we found empirical evidence for a domino effect in Windows Server 2003. Defects in one component can significantly increase the likelihood of defects (in other words the probability of defects) in dependent components. This is a significant issue in understanding the cause-effect relationship of defects and the potential risk of propagating a defect through the entire system.

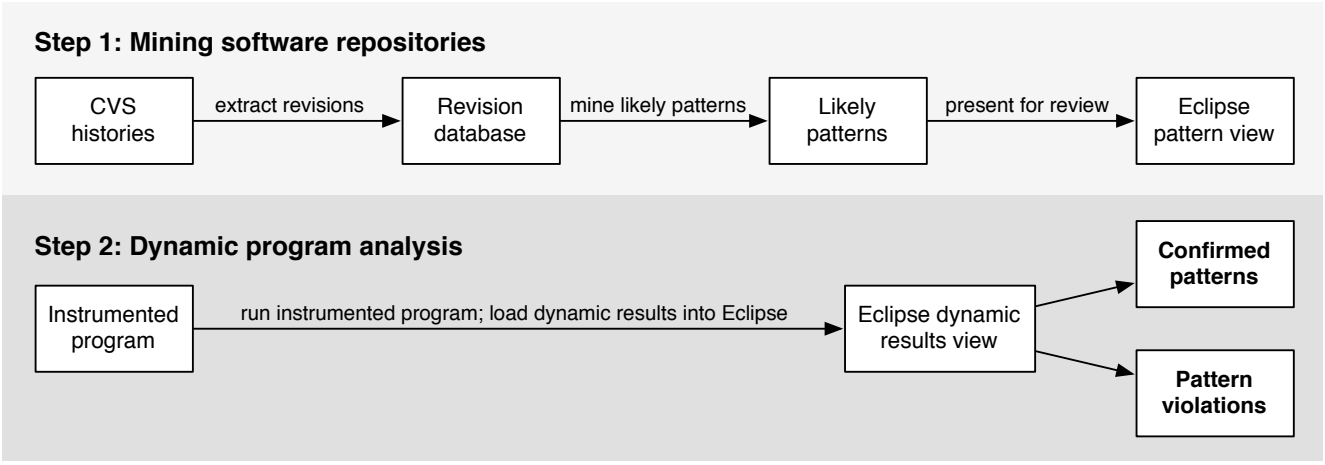

Figure 2. Architecture of DYNAMINE. The first row represents revision history mining. The second row represents dynamic analysis.

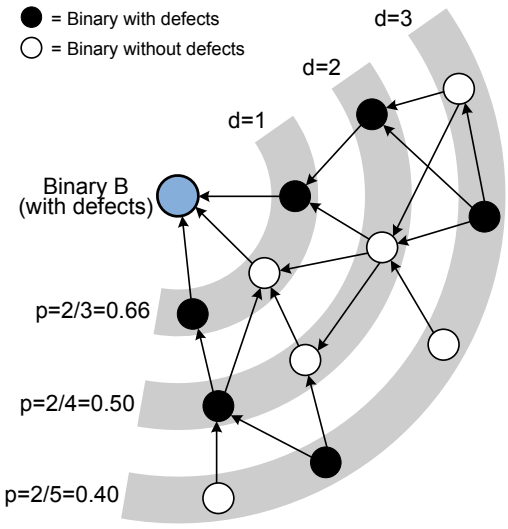

Figure 3. Illustrative example of a domino effect. The defect probability decreases as the distance to a defect-prone binary increases.

5 Contributions

Software development results in a huge amount of data: changes to source code are recorded in version archives, bugs are reported to issue tracking systems, and communications are archived in e-mails and newsgroups. Mining software repositories makes use all of this data to understand and support software development. This dissertation makes the following contributions to this area.

Fine-grained analysis of version archives. The work on DYNAMINE was the first to analyze particular code changes and not only the changed location. DYNAMINE learned *project-specific usage pattern of methods* from version archives and validated the patterns with dynamic program analysis, which is another novelty.

The aspect-mining tool HAM reveals *cross-cutting changes*: "A developer invoked lock() and unlock() in 1,284 different locations." In aspect-oriented programming, such changes can be encapsulated as aspects. By breaking down large code-bases into their evolution steps, HAM scales to large systems such as Eclipse.

Mining bug databases to predict defects. In software development, the resources for quality assurance (QA) are typically limited. A common practice among managers is *resource allocation* that is to direct the QA effort to those parts of a system that are expected to have most defects.

This dissertation presented techniques to build models that can successfully predict the most defect-prone parts of large-scale industrial software, in our experiments Windows Server 2003. The proposed measures on dependency graphs outperformed traditional complexity metrics. In addition, we found empirical evidence for a domino effect: *depending on defect-prone binaries increases the chances of having defects.*

Dependencies between subsystems are typically defined early in the design phase; thus, designers can easily explore and assess design alternatives in terms of expected quality.

6. Mining Repositories Across Projects

Mining software repositories works best on large projects with a long and rich development history; smaller and new projects, however, rarely have enough data for the above techniques. Our future work, will therefore focus on *mining software repositories across projects*. We hypothesize that projects which do not have enough history can learn from the repositories shared by other similar projects. For instance, open-source communities (such as SourceForge.net) host several thousand projects, which are all available for mining. Similarly, within an industrial setting, companies can learn from all their ongoing and completed projects.

Having access to the history of other projects supports developers and managers to make well-informed decisions, for instance with respect to design ("Which library should we use?"), personnel ("Who is qualified for this task?"), and resource allocation ("What parts should we test most?). They can identify similar situations in the past, and see how these situations impacted the evolution of a project. Overall, the goal is to automate most of this process and provide appropriate tool support for both open- and closed-source software development.

On the one hand, we expect that existing mining techniques will benefit from a *larger population* of projects. For instance, change classification frequently finds insufficient evidence within a single project to blame bad changes, which results in a large number of false negatives [6]. By extending the search space to many projects, we are more likely to find enough evidence. We can also *transfer knowledge* from one project to another similar project. Nagappan et al. [8] observed that defect prediction models trained on one project can reliably predict defects for projects with comparable development processes. On the other hand, having access to many projects poses new research questions, one of them being: *"What can we mine from such data in an automatic, large-scale (many projects), and tool-oriented fashion to support software development?"*

At the beginning of the last century, the philosopher George Santayana remarked that those who could not remember the past would be condemned to repeat it. In other words, to achieve progress, we must learn from history. With our future research, everyone will get enough history from which to learn.

To learn more about our research, we invite you to visit

`http://www.softwaremining.org/`

References

[1] S. Breu and T. Zimmermann. Mining aspects from version history. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, 2006.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[3] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[4] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *VLHCC'05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, 2005.

[5] H. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.

[6] S. Kim, K. Pan, and E. J. Whitehead, Jr. Memories of bug fixes. In *SIGSOFT'06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 35–45, 2006.

[7] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 296–305, 2005.

[8] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE'06: Proceeding of the 28th international conference on Software engineering*, pages 452–461, 2006.

[9] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):221–232, 1975.

[10] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting navigation in software maintenance. In *ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, 2005.

[11] G. Venolia. Textual alusions to artifacts in software-related repositories. In *MSR'06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 151–154, May 2006.

[12] T. Zimmermann and N. Nagappan. Predicting subsystem failures using dependency graph complexities. In *ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability*, pages 227–236, 2007.

[13] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 531–540, 2008.

[14] 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.