Learning from Project History to Support Programmers

Thomas Zimmermann

Graduiertenkolleg „Leistungsgarantien für Rechnersysteme“
Fachrichtung 6.2 – Informatik
Universität des Saarlandes
66041 Saarbrücken
zimmerth@cs.uni-sb.de

Abstract: Nowadays, any larger software project collects lots of data. Changes to source code are stored in version control archives, and problems in bug databases. In this paper we present two tools that mine both sources to help developers writing better programs. The eROSE tool guides programmers along related changes in a similar way Amazon.com supports its customers: You changed fKeys[] and eROSE suggests initDefaults(), because in the past both items always have been changed together. The HATARI tool warns programmers against risky locations: “Programmers failed at changing resolveClasspath(). Be careful.” To measure risk, HATARI locates changes that lead to problems, as indicated by later fixes.

1 Introduction

Shopping for a book at Amazon.com, you may have come across a section that reads “Customers who bought this book also bought…”, listing other books that were typically included in the same purchase. Such information is gathered automatically by mining the purchase history of Amazon.com.

We developed a similar feature for software development: “Programmers who changed these functions also changed…” Our eROSE tool mines this information automatically from version histories and guides programmers along related changes (Section 2).

We also leveraged bug databases to identify (past) changes that resulted in later fixes. Since such changes are not beneficial, we use them to compute risk values for individual locations. Our HATARI tool makes developers aware of risky locations (Section 3).

2 eROSE: Guiding Programmers

Our eROSE tool learns from the information stored in version archives to make recommendations for programmers. These recommendations are useful in two common scenarios:

- **The "Improve Navigation" scenario.** A major application for eROSE is to guide users through source code: The user changes some entity and eROSE automatically recommends related changes in a view. Figure 1 shows our eROSE tool as a plug-in for the ECLIPSE programming environment. The programmer is inserting a new preference, and has added an element to the fKeys[] array. eROSE now suggests to consider further changes, as inferred from the version history. First come the locations with the highest *confidence*—that is, the likelihood that further changes be applied to the given location. Position 3 on the list is an HTML documentation file with a confidence of 0.727—suggesting that after adding the new preference, the documentation should be updated, too. Such a dependency is undetectable by program analysis.

- **The "Prevent Errors" scenario.** Besides supporting navigation, eROSE also *prevents errors*. The idea is that when a user decides to commit changes to the version archive, eROSE checks if there are related changes with a *high confidence* that have not been changed yet. If there are, like in Figure 1 the top two locations, eROSE issues a pop-up window with a warning. It also suggests the "missing" items that should be considered.

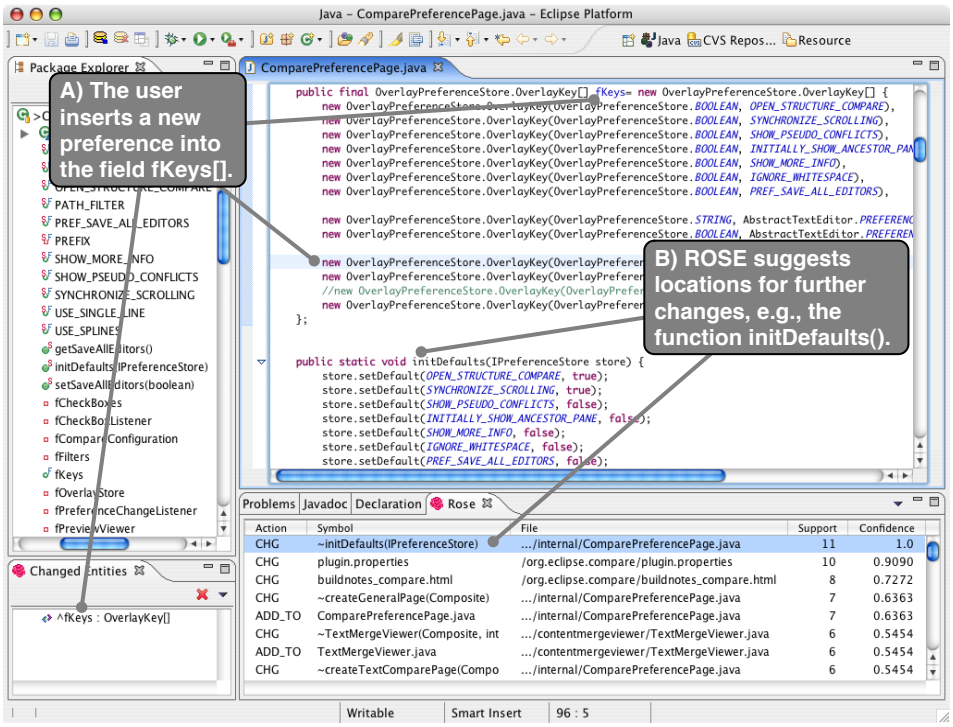

Figure 1: After the programmer has made some changes to the source (above), eROSE suggests locations (below) where, in similar transactions in the past, further changes were made.

Each recommendation from Figure 1 corresponds to an association rule. Such rules are frequently used in data mining to represent patterns. As an additional example consider the following rule

{ *alter*(fKeys[]) , *alter*(initDefaults()) } ⇒ { *alter*(plugin.properies) }
[support count=10; confidence=0.9090]

This rule represents the situation that a developer has altered *both* entities fKeys[] and initDefaults().[1] Based on this situation, eROSE recommends to alter plugin.properties because 90.90% of all changes for both fKeys[] and initDefaults() have also altered this file in the past. These 90.90% correspond to a total of 10 changes.

In contrast to classical association rule mining with the Apriori algorithm [RR94], eROSE does not compute all rules beforehand. Instead it uses optimizations that allow the efficient computation of rules *on demand*. For more information about this and a detailed evaluation of eROSE we refer to our previous work [ZWDZ05].

3 HATARI: Raising Risk Awareness

Developers frequently change software in order to improve quality. Unfortunately, not all changes are beneficial. Any bug database will show a significant fraction of problems that are reported some time after some change has been made.

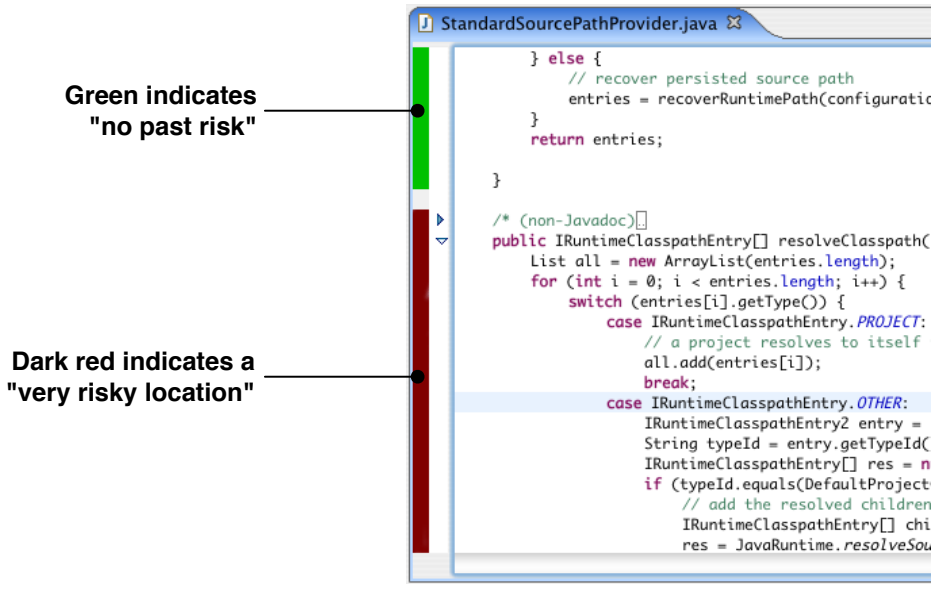

Figure 2: HATARI annotates source code with risk bars. A dark red bar reads as "Be careful. Programmers failed at changing." A green bar tells that no risky changes have been observed (yet).

[1] In addition to *alter* changes, eROSE takes *addition* and *deletion* of entities into account.

When it comes to determining the risk of a change inducing a later problem, the *location* of the change is a significant factor. Our HATARI prototype makes this risk visible for developers by annotating source code with color bars. A dark red annotation like in Figure 2 reads as "Programmers failed at changing resolveClasspath(). Be careful." In contrast, a green annotation tells that no risky changes have been observed (so far). Furthermore, HATARI provides views to browse through the most risky locations and to analyze the risk history of a particular location.

HATARI determines the risk of locations automatically from history—in particular, the project's version archive and the project's bug database:

1. HATARI starts with a bug report in the bug database, indicating a *fixed problem*. It extracts the associated change from the version archive, giving us the *location* of the fix.

2. HATARI determines the *earlier change* at this location that was applied before the problem was reported. This earlier change is the one that *caused* the later fix, which is why we call it *fix-inducing*.

3. For each location, HATARI determines all changes that were ever applied to the location, and computes the individual *risk of change* as a percentage of fix-inducing changes.

We refer to our previous work for more information and case studies on fix-inducing changes [SZZ05b] and a detailed description of HATARI [SZZ05a].

4 Acknowledgements

This project is funded in part by the Deutsche Forschungsgemeinschaft under the grant Ze 509/1-1. The author is supported by a research fellowship of the Graduiertenkolleg "Leistungsgarantien für Rechnersysteme". Thanks to Michael Burch, Stephan Diehl, Peter Weißgerber, and Andreas Zeller for their discussions and comments.

References

[SZZ05a] J. Sliwerski, T. Zimmermann, A. Zeller: *HATARI: Raising Risk Awareness.* Demo Paper. Proc. European Software Engineering Conference/ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Lisbon, Portugal, Sept. 2005.

[SZZ05b] J. Sliwerski, T. Zimmermann, A. Zeller: *When do Changes Induce Fixes? On Fridays.* Proc. International Workshop on Mining Software Repositories (MSR), Saint Louis, Missouri, USA, May 2005, pages 24–28.

[ZWDZ05] T. Zimmermann, P. Weißgerber, S. Diehl, A. Zeller: *Mining Version Histories to Guide Software Changes.* IEEE Transactions on Software Engineering, 31(6), 2005.

[RR94] R. Agrawal, R. Srikant: *Fast Algorithms for Mining Association Rules in Large Databases*. Proc. International Conference on Very Large Data Bases (VLDB), Santiago de Chile, Chile, Sept. 1994, pages 487–499.