HATARI: Raising Risk Awareness

Jacek Śliwerski
International Max Planck Research School
Max-Planck-Institut für Informatik
Saarbrücken, Germany
sliwers@mpi-sb.mpg.de

Thomas Zimmermann · Andreas Zeller
Department of Computer Science
Saarland University
Saarbrücken, Germany
{tz, zeller}@acm.org

ABSTRACT

As a software system evolves, programmers make changes which sometimes lead to problems. The risk of later problems significantly depends on the location of the change. Which are the locations where changes impose the greatest risk? Our HATARI prototype relates a version history (such as CVS) to a bug database (such as BUGZILLA) to detect those locations where changes have been risky in the past. HATARI makes this risk visible for developers by annotating source code with color bars. Furthermore, HATARI provides views to browse through the most risky locations and to analyze the risk history of a particular location.

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

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.

When it comes to determining the risk of a change inducing a later problem, the *location* of the change is a significant factor. In this work, we attempt to *identify the individual risk for all code locations*—by examining, for each location, whether earlier changes caused problems. Our HATARI[1] prototype makes this risk visible for developers by annotating source code with color bars. Furthermore, HATARI provides views to browse through the most risky locations and to analyze the risk history of a particular location.

[1]*Hatari* is the Swahili word for "risk" or "danger".

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

HATARI determines the risk of locations automatically from project artifacts—in particular, the project's *version archive* (such as CVS) and the project's *bug database* (such as BUGZILLA):

1. HATARI starts with a bug report in the bug database, indicating a *fixed problem.* HATARI 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.

Why would one want to know about the past risk of changes?

- Locations that are risky to change are typical candidates for *maintenance*, such as extra documentation or restructuring.
- When it comes to *quality assurance*, changes that occur at risky locations should get more attention in changing, testing, and reviewing than changes at "safe" places.

In this paper, we demonstrate how HATARI determines risky locations and makes them usable to the programmer. We also present benefits for the user, and briefly discuss how HATARI advances beyond the state of the art.

2. RISKY LOCATIONS

Every programmer knows that there are locations in the code where it is difficult to get things right. As an example, suppose you are a programmer working on the ECLIPSE source code; your task is to change the function `resolveClasspath()` in Figure 1. At a first glance, `resolveClasspath()` looks like any other function and should not be too difficult to change. However, its change history tells us a different story:

Revision 1.3 fixed bug 16313, *"NPE out of StandardSourcePathProvider"*, and added a null pointer check to the function to avoid NullPointerExceptions (NPE).

Revision 1.4 fixed bug 7999, *"Source lookup with Runtime JRE"*, and deleted the complete function `resolveClasspath()`.

Revision 1.5 fixed bug 26681, *"Multiple output folder"*, and added a new implementation of `resolveClasspath()`. This implementation was not perfect, it was revised later in revision 1.7 to deal with several kinds of classpath entries.

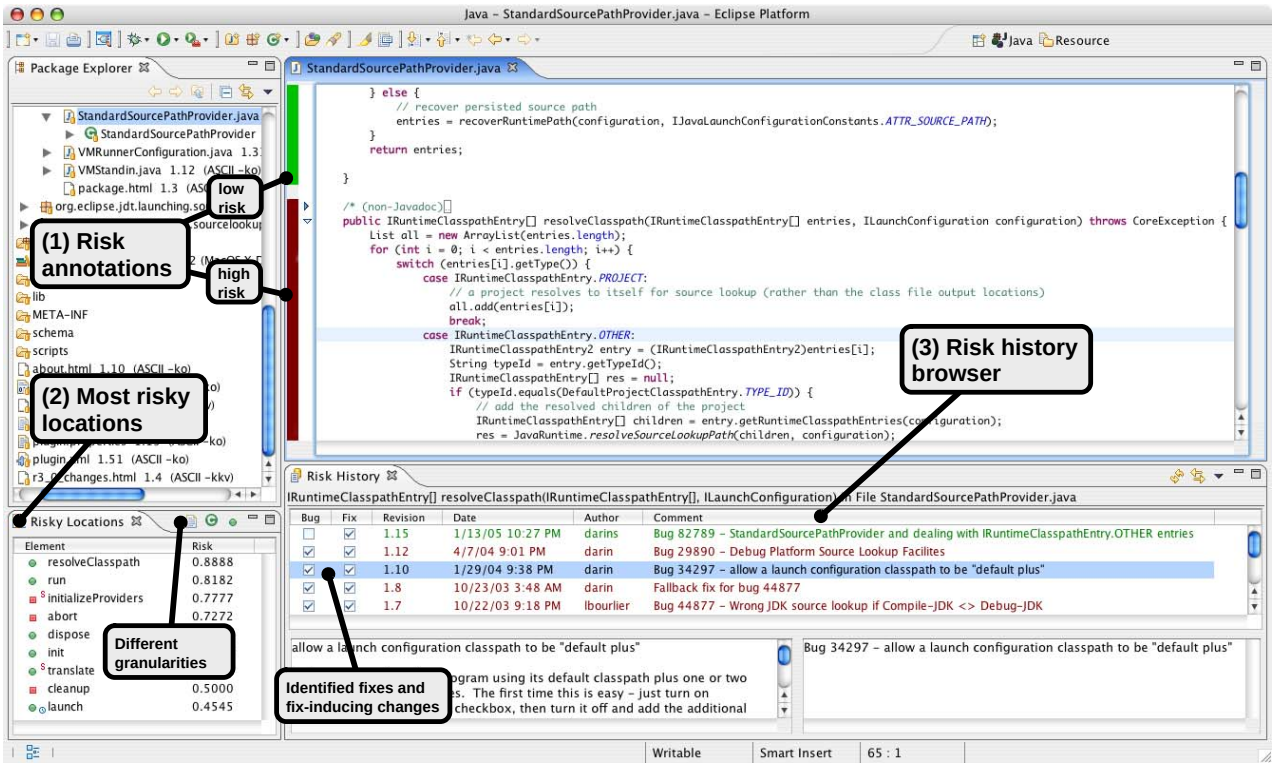

Figure 1: The HATARI plug-in for ECLIPSE annotates source code with colors that indicate risk (1). For maintenance, developers can browse through all risky locations (2) or investigate the risk history of particular locations (3).

Revision 1.7 tried to fixed bug 44877, *"Wrong JDK source lookup if Compile-JDK <> Debug-JDK"*, by adding code for a new entry `RuntimeClasspathEntry.CONTAINER`.

Revision 1.8 undid the previous fix for bug 44877 in revision 1.7.

Revision 1.10 fixed bug 34297, *"allow a launch configuration classpath to be 'default plus"'*, by adding code for a new entry `IRuntimeClasspathEntry.OTHER`.

Revision 1.12 fixed bug 29890, *"Debug Platform Source Lookup Facilites"* and revised the change in 1.10 to deal with new cases and added a missing null pointer check.

To cut a long story short, *all changes* to `resolveClasspath()` resulted in later fixes. And the story still goes on: revision 1.15 added a missing else block that was forgotten in revision 1.12.

Obviously, `resolveClasspath()` is a function that is difficult to get right—it was changed 9 times, and all of these changes were fixes. Worse even, 8 out of these 9 fixes directly resulted in an externally visible problem, as documented in the ECLIPSE bug database. This means that `resolveClasspath()` is *risky to change*—almost all people who tried got it wrong.[2]

At this point, note that "risky to change" is different from "frequently fixed". Although `resolveClasspath()` is both frequently fixed as well as risky to change, there are locations A where changes frequently induce fixes in other locations B. In such cases, the locations B are frequently fixed. However, blaming B for the problems would be barking at the wrong tree: Fixes and problems were induced by the changes in A, which is thus risky to change.

By relating the change history to the bug database, we can locate this risk of change—and make it visible to developers. Our HATARI prototype annotates locations with color bars that indicate their risk (see Figure 1). In addition, HATARI provides views to browse through the most risky locations and to analyze the risk history of a particular location. In the next sections, we will describe HATARI's integration into ECLIPSE and its underlying techniques.

[2]In fact, `resolveClasspath()` holds the record for being the most risky location in ECLIPSE.

3. HATARI'S ECLIPSE INTEGRATION

Our HATARI prototype integrates risk information into ECLIPSE in three ways (see Figure 1):

Annotating locations. For each location HATARI measures its past risk and displays it as a colored box on the left side of the editor. Inspired by the Emerald tool [4], we use a scale of green and different shades of red to visualize the risk of change at a location. These *annotations* are intended to raise the risk awareness among developers when they make changes.

Risk history view. If a developer clicks on an annotation, the *risk history view* is opened for the location. This view contains information about past fixes, fix-inducing-changes, and regular changes. Furthermore, it has functionality to compare different revisions, so that developers can reconstruct the risk history of a location (like we did in Section 2). The main use of this view is for maintenance, e.g., to find out why a particular location is risky.

Risky locations view. Another tool for maintenance is the *risky locations view* which contains a ranking of all locations based on their risk values.

All that HATARI needs to make it work is a version archive (such as CVS) and a problem archive (such as BUGZILLA).

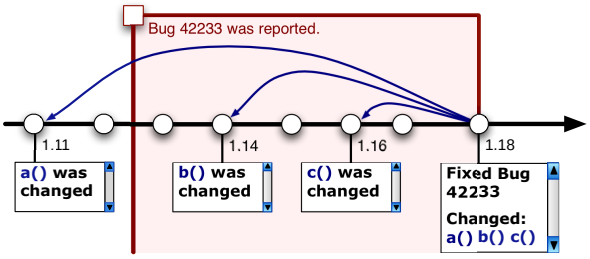

Figure 2: Locating fix-inducing changes for bug 42233

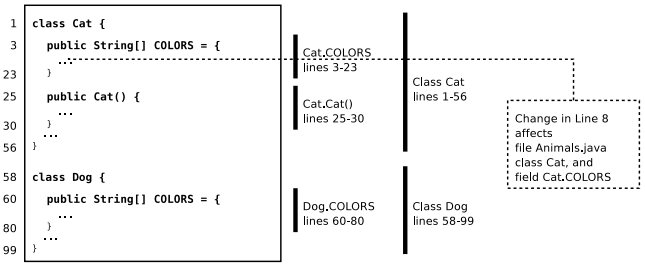

Figure 3: Relating changes to locations.

4. HOW HATARI WORKS

Let us now briefly discuss how HATARI obtains the risk information. HATARI proceeds in four automatic steps:

1. HATARI identifies the fixes made to a software system.
2. For each of these fixes, HATARI identifies the *fix-inducing* change(s) that last touched the fixed locations.
3. For each of the fix-inducing changes, HATARI determines their location.
4. Finally, for every location in the source code, HATARI measures their risk of change.

The following sections provide details on these steps.

4.1 Identifying Fixes

In order to locate fix-inducing changes, HATARI first needs to know if a change is a fix. While advanced version control systems allow the programmer to specify the nature of a change, CVS has no such feature. Therefore, we identify fixes based on the *log message* that is supplied with a change. Specifically, HATARI looks for *keywords* such as "Fix", as introduced by Mockus and Votta [5] as well as for *references to bug databases*, as introduced by Fischer et al. [3] as well as Čubranić and Murphy [2]. In Figure 2, both approaches would recognize revision 1.18 as a fix, because of the keywords "Fixed" and "Bug" and because of a bug identifier in the bug database "42233".

In addition to these techniques, we also map *user information* from version archives to problem archives and vice versa; the basic idea is that whenever a problem is marked as "closed" in the bug database, the most recent change made by this developer is likely to be the fix that closed the problem [9].

Formally, let B be the set of all bug reports; the set of all changes is C. HATARI establishes links (b, c) between a bug report b and a fix c and collects such links in a *bug-change* relation $L \subseteq B \times C$.

4.2 Locating Fix-Inducing Changes

After HATARI has isolated fixes, it determines *the earlier changes* which caused these fixes. We call such changes *fix-inducing.* Note that a fix-inducing change also induces the problem which lead to the fix. Although we'd like to know about problem-inducing changes, we can only identify the change which lead to the problem once we have the fix.

Once more, consider the example in Figure 2, where HATARI has identified the change in revision 1.18 to be a fix. HATARI first computes the differences between the earlier revision (1.17) and the fixed revision (1.18). As a result it gets the lines that have been changed by the fix.

Next, HATARI annotates each line of the earlier revision (1.17) with the most recent author and revision that touched this line. Right now, we use the CVS *annotate* command, but it is straightforward to implement a similar feature for other version control systems.

HATARI uses these annotations and the set of changed lines to find *candidates* for fix-inducing changes. For our example, we assume that lines 20, 40, and 60 have been changed; thus our candidates are the changes leading to revisions 1.11, 1.14, and 1.16.

Finally, HATARI uses the bug database to rule out changes that cannot be fix-inducing because they have been made after the bug was reported, i.e., they cannot be real causes for the bug. In our example, revisions 1.14 and 1.16 are not fix-inducing. Without the connection to the bug database, they would be marked fix-inducing and therefore be false positives.

In our example, revision 1.11 is the only fix-inducing change. Frequently, but not always, such fix-inducing changes introduced the bug that has been fixed later on. However, such changes are always unstable and thus risky.

Formally, we define a *induced-change* relation $J \subseteq C \times C$ that connects two changes c_i and c_j with each other if and only if c_j changed a line that was introduced in c_i; in terms of CVS this means that c_i is included in the annotations of c_{j-1} for the lines changed by c_j. The induced-change relation can be build for any changes, regardless whether c_j is a fix or not.

For fix-inducing changes, we combine the bug-change relation L and the induced-change relation J. Every change c_d that is later undone by a fix c_f for a bug report b, is fix-inducing, if b has been reported after c_d was performed. Thus the set of fix-inducing changes F is defined as:

$$F = \{c_d \mid \exists\,(c_d, c_f) \in \text{ induced-change relation } J, \\ \exists\,(b, c_f) \in \text{ bug-change relation } L, \\ \textit{timestamp}(c_d) < \textit{timestamp}(b) < \textit{timestamp}(c_f)\}$$

Note that fix-inducing changes are not known as fix-inducing when they are made. They can only be identified as fix-inducing when the corresponding fix is made.

4.3 Assigning Locations to Changes

Before HATARI can investigate the risk of locations, it has to assign changes to locations. Currently, HATARI supports three different kinds of locations:

File granularity. We get the affected file directly from the fix-inducing changes. For this granularity we need no additional preprocessing.

Class granularity. We assign fix-inducing changes to classes by mapping the lines changed by a fix to the surrounding class.

Member granularity. We assign fix-inducing changes to members of classes, such as methods and fields, by mapping the lines changed by a fix to the surrounding member.

Zimmermann and Weißgerber describe how to map changed lines to locations like classes and their members [10]. The idea is sketched

in Figure 3, where the change in line 8 affects the class `Cat` and the field `Cat.COLORS`. In the case of HATARI, we can additionally assume that a fix-inducing change is in the same location as its corresponding fix. This reduces the number of revisions that HATARI has to parse because for *one* fix there may be *several* fix-inducing changes. Using the mapping for fixes, HATARI assigns to each fix-inducing change the touched locations.

4.4 Measuring Past Risk

Finally, HATARI has to determine the risk of each location in the source code. We define the *past risk* of a location e as the likelihood that a change made to this location was later undone by a fix.

$$risk_{past}(e) = \frac{|\{c_b \mid c_b \in F, c_b \text{ changed } e\}|}{|\{c \mid c \in C, c \text{ changed } e\}|}$$

The motivation behind this definition is that unrisky changes are never touched again or only by changes that introduce new features. As mentioned before, we can easily assign changes to locations.

Finally, HATARI visualizes this risk for each location, and ranks the locations according to their risk of change, as shown in Figure 1.

5. PRACTICAL BENEFITS

What does the usage of HATARI mean for the programmer? To start with, HATARI is helpful to *make choices:* If some task can be realized in a number of ways, the risk of change at different locations can be an important factor to consider.

Just as in real life, we cannot always avoid risk. However, knowing about risk, we can take appropriate *precautions.* If a change has to be made at a risky location, the programmer can examine the past history, and learn why earlier changes have lead to problems. Furthermore, a change to a risky location means that extra effort should be spent on reducing the risk; for instance, changes made to risky locations may be specially reviewed or tested.

Finally, a high risk of change is an indicator of high *factual complexity:* How hard is it getting things right? Locations with a high risk are candidates for measures that reduce complexity, such as restructuring or better documentation. Furthermore, risk as an indicator for complexity can help in determining correlated code features, or in calibrating software metrics.

6. RELATED WORK

HATARI is the first tool to use past risk as an indicator for future risk, but it is not the first tool to correlate bug reports and version histories.

The work closest to ours in spirit is the paper "Predicting Risk of Software Changes" by Mockus and Weiss [6]. Just like Mockus and Weiss' work, we focus on predicting the risk of software changes. However, we rely on the past risk of change to predict the future risk—rather than relying on indicators like "frequently fixed" or "size of change". Furthermore, HATARI predicts the risk uniquely from the location the change is applied to, and therefore is able to predict the risk even before an actual change is made—which allows for identifying those locations where change equals risk.

Several researchers in the past have combined problem and version archives to search for locations which are likely to be defective [8, 7]. In contrast to all these approaches, HATARI does not aim to predict the number of remaining defects, but the risk of *introducing a new defect when making a change.* Furthermore, HATARI relies on a single, yet highly relevant factor—the risk in the past.

The concept of fix-inducing changes was introduced by Śliwerski et al. who also presented initial mining results [9]. The similar concept of *fix-on-fix changes* was originally proposed by Baker and Eick [1]. However, while fix-on-fix changes focus on the induced fix, HATARI investigates the original fix-inducing change.

7. CONCLUSION AND CONSEQUENCES

If a code location has caused problems in the past, it is likely to do so in the future. HATARI identifies and locates this risk. Locations for which HATARI determines a high risk should be subject to increased quality assurance, as well as possible restructuring.

On the practical side, HATARI can easily be plugged into existing projects. In particular, it needs no program analysis or other knowledge about the process and the artifacts; all that is needed is a version archive and a bug database. To make the results available to programmers and managers, we turned HATARI into an ECLIPSE plug-in that makes the risk of individual code changes visible to the programmer.

By clicking on a shade, the programmer can explore the past history, thus learning the earlier changes and induced problems that occurred at that location. This is a benefit of HATARI: By relying on one single factor (past risk), it brings much better explanations than multi-factor metrics which predict risk by curve fitting.

For ongoing information on the HATARI project, see

`http://www.st.cs.uni-sb.de/softevo/`

Acknowledgments. The HATARI project is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Valentin Dallmeier and Stephan Neuhaus provided valuable comments on earlier revisions of this paper.

8. REFERENCES

[1] M. J. Baker and S. G. Eick. Visualizing software systems. In *Proceedings of the 16th International Conference on Software Engineering*, pages 59–70. IEEE Computer Society Press, May 1994.

[2] 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.

[3] 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 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[4] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. Emerald: Software metrics and models on the desktop. *IEEE Software*, 13(5):56–60, 1996.

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

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

[7] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering (ICSE)*, May 2005.

[8] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, New York, NY, USA, 2004. ACM Press.

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

[10] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Edinburgh, Scotland, UK, May 2004.