

Lehrstuhl für Softwaretechnik
Universität des Saarlandes

Master Thesis
Locating the Risk of Changes

Jacek Śliwerski
sliwers@mpi-inf.mpg.de

**Advisors: Prof. Dr. Andreas Zeller
Thomas Zimmermann**

Saarbrücken, 2005

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?

We introduce a set of automated techniques that relate a version history archive (such as CVS) with a bug database (such as BUGZILLA) to detect those locations where changes have been risky in the past. Our experiments show that simple measures have low accuracy in locating files that are most risky to change.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Motivation	2
1.3	Contributions	3
1.4	Structure	4
2	Essentials	5
2.1	Quick Tour	5
2.2	Version Control Systems	6
2.3	Bug Tracking Systems	8
2.4	Summary	9
3	Linking Transactions to Bugs	11
3.1	Syntactic Analysis	12
3.2	Semantic Analysis	13
3.3	Quality	14
3.4	Summary	16
4	Locating Fix-Inducing Changes	17
4.1	Example	17
4.2	Formal Description	19
4.3	Results	20
4.3.1	Fix-Inducing Transactions are Large	20
4.3.2	Do not Program on Fridays	21
4.4	Summary	22
5	Evaluation	25
5.1	Evaluation Setup	26
5.2	Data Collection	28
5.3	Experiment Description	29
5.4	Results	32

5.5	Correlation	33
5.6	Discussion	37
5.7	Summary	37
6	Related Work	39
6.1	Software Repositories and Bug Tracking Systems	39
6.2	Defect Prediction	40
7	Conclusions	41
A	Tools	43
A.1	Preprocessing	43
A.2	Prediction	45
A.2.1	Collecting Predictions	45
A.2.2	Analysis	46
	Bibliography	47

List of Figures

1.1	Example usage of our approach	2
2.1	General idea of our approach	6
2.2	Simplified scheme of our database	7
3.1	Linking transactions with bug reports	11
4.1	Locating fix-inducing changes	18
4.2	CVS annotation example	18
5.1	Experiment scheme	30
5.2	Experiment results (ECLIPSE, non-relative)	34
5.3	Experiment results (ECLIPSE, relative)	34
5.4	Experiment results (MOZILLA, non-relative)	35
5.5	Experiment results (MOZILLA, relative)	35
5.6	Experiment results (COLUMBA, non-relative)	36
5.7	Experiment results (COLUMBA, relative)	36
A.1	Example implementation of the <code>cvsann</code> tool	44
A.2	Example call of the <code>cvsbts</code> tool	45
A.3	Example call of the <code>Errasmus</code> tool	46
A.4	Example call of the <code>Analysis</code> tool	46

List of Tables

3.1	Distribution of links in ECLIPSE	15
3.2	Distribution of links in MOZILLA	15
3.3	Manual classification breakdown	16
3.4	Classification correspondence	16
4.1	Algorithm for finding fix-inducing changes	19
4.2	Transaction sizes in ECLIPSE	20
4.3	Transaction sizes in MOZILLA	21
4.4	Day-of-week breakdown for ECLIPSE	23
4.5	Day-of-week breakdown for MOZILLA	23
5.1	List of predictors	26
5.2	Summary of projects used in the experiment	29
5.3	Experiment results (mixed-granularity level, April 1, 2003) . .	33
5.4	Correlation coefficients for file-granularity level (April 1, 2003)	33

Chapter 1

Introduction

Developers frequently change software in order to improve its 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 of change for all code locations*—by examining, for each location, how many earlier changes caused problems. We present a set of automated techniques that determine the risk of locations solely from project artifacts—namely the project’s *bug database* (such as BUGZILLA) and *version archive* (such as CVS).

1.1 Overview

Our approach consists of the following steps:

1. Start with a bug report in the bug database indicating a *fixed problem*.
2. Extract the associated change from the version archive, thus giving the *location* of the fix.
3. Determine 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*.
4. For each location, determine the changes that were ever applied to it and compute the individual *risk of change* as the percentage of fix-inducing changes.

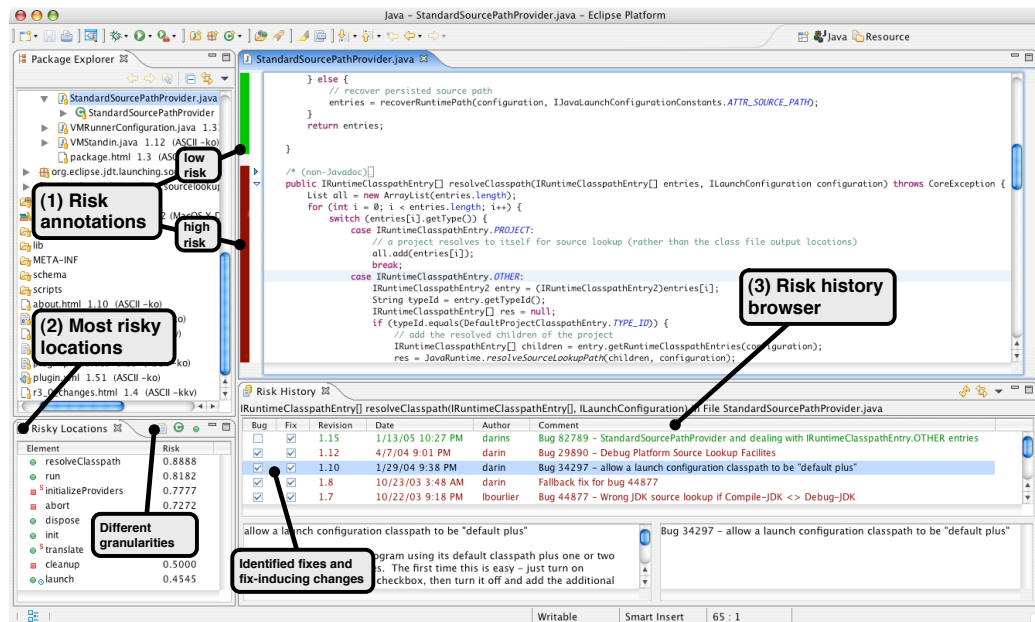


Figure 1.1: The HATARI plug-in for ECLIPSE annotates source code with colors that indicate risk (1). For maintenance, developers can browse through all risky locations (2) or investigate the risk history of particular locations (3)

1.2 Motivation

Why would one want to know about the past risk of changes? The general idea is that *past risk is an indicator for future risk*, our main hypothesis that we test later on:

- 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 testing and reviewing than changes that occur at “safe” places.
- The risk of change can be seen as a *factual complexity measure*: How hard is it getting things right? Risk can thus help in determining correlated code features, or in calibrating software metrics.

This approach may be turned into a tool that indicates riskiness of the location such as the HATARI plug-in for ECLIPSE developed at the Saarland University (see Figure 1.1). Information about the risk is conveyed to a developer by the means of:

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 [HAK⁺96], 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. 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.

1.3 Contributions

This thesis is not the first attempt to identify risk in software projects. In earlier work researchers were looking for a correlation between size of the file and defect density [FO99], combining various file properties in order to find most frequently fixed locations [OWB04] and inspecting the risk of introducing an individual change [MW00]. In contrast to the state of the art, the presented work:

- introduces the concept of *fix-inducing changes* and new risk predictors based on this notion,
- uses the new notion of *risk* expressed as a percentage of fix-inducing changes in the history of a location,
- compares accuracy of several simple measures in predicting the risk of future changes.

For a full discussion of related work see Chapter 6. Parts of this thesis have been published and presented at

- the *International Workshop on Mining Software Repositories* (MSR 2005) [SZZ05b],
- the *ACM Sigsoft Symposium on the Foundations of Software Engineering* (FSE 2005) [SZZ05a].

1.4 Structure

The remainder of this thesis is organized as follows: Chapter 2 gives an *overview* of our approach and sketches some necessary, preliminary techniques that are not subject of this thesis. Chapter 3 describes how we locate the changes where a bug fix has been applied. Chapter 4 presents the algorithm for finding *fix-inducing changes*. Chapter 5 covers the *evaluation* of our approach. Chapter 6 gives an overview of *related work* and Chapter 7 concludes with the presentation of *future work*. Finally, Appendix A describes the *tools* created in order to cover the needs of this thesis.

Chapter 2

Essentials

This chapter gives an insight into our approach and describes both version control systems and bug tracking systems—two sources of historical data about the project that we rely on.

2.1 Quick Tour

Our approach consists of two main parts: preprocessing and prediction. During the preprocessing phase, we build a database which is a source of extensive knowledge about the history of every file, function, class, module, etc. This knowledge is used to predict the risk of change in the second phase. Preprocessing and prediction consist of several smaller steps, as presented in Figure 2.1:

1. Mapping Project's History

In the first step we mirror a *bug tracking system* and a *version archive* to a common database. Short descriptions of both data sources and mapping procedures are given in Sections 2.2 and 2.3 below.

2. Locating Fix-Inducing Changes

- (a) Next, we extend the database in such a way that it includes information about the purpose of the change. This step is described in detail in Chapter 3.
- (b) As the last step of the preprocessing phase we insert records that indicate whether a change is fix-inducing or not. For further information see Chapter 4.

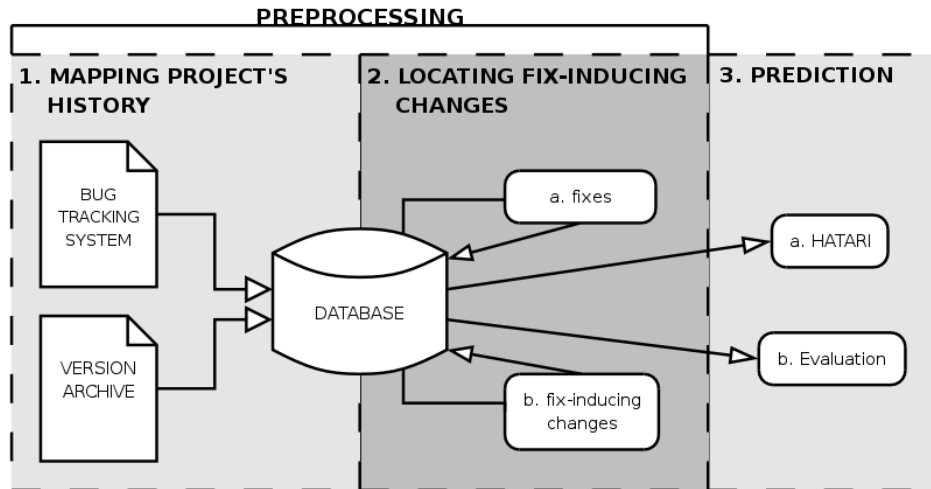


Figure 2.1: General idea of our approach

A simplified scheme of the complete database is presented in Figure 2.2. It is composed of four parts: a *version control archive* data (left pane), bugs stored in a *problem tracking system* (right pane), links between *transactions* and fixed *bugs* and information about fix-inducing entities.

3. Prediction

- (a) HATARI queries the database to retrieve the history of a particular entity. As a result, it receives a list of its changes with information whether they are fix-inducing or not. A detailed description of how HATARI works may be found in [SZZ05a].
- (b) To assess the usability of our approach we query the database for histories of all entities from the repository and split them into two parts: *past* and *future*. We feed several simple measures with past information and compare the outcomes with the information about the future. The evaluation is addressed in Chapter 5.

2.2 Version Control Systems

Version control systems have been developed to enable concurrent work of multiple developers on the same project. In most common version control systems, there exists a central repository which stores the current (official) version of the sources. A developer checks out files from the repository and works on her local copy. Once finished with modifications, she commits her

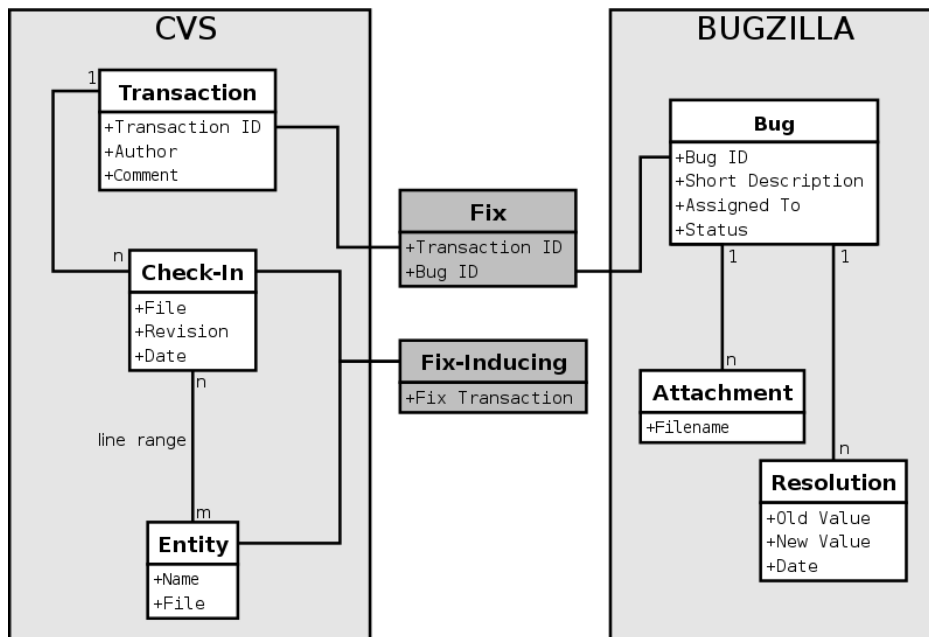


Figure 2.2: Simplified scheme of our database

changes back to the repository, attaching a short message describing what has been done. Version control systems offer their assistance in merging multiple, concurrent changes to the same file as well as in resolving conflicts between them. But, most importantly, version control systems enable developers to access any version of their files. The most popular version control systems are CVS, SUBVERSION and SCCS.

Mirroring the CVS data to the database is based on the work by Zimmermann and Weißgerber [ZW04] and is similar to the work by Fischer, Pinzger and Gall [FPG03b]. The extraction tool not only extracts the change data from the repository, but also makes further preprocessing. The resulting part of the database is shown in the left pane of Figure 2.2 and contains, among others, the following information:

Check-In. When a developer commits changes in a *file* to the repository, she creates a new *revision* of this *file*. Such an operation (called *check-in*) is stored in our database with a precise *time* when it has been made.

Transaction. Check-ins are grouped into one *transaction* if they have been all committed to the repository by the same *developer*, with the same *log message*, within a small time-frame. We use *sliding window approach* which, in contrast to *fixed window approach*, shifts the time-frame to the last grouped change until no other change fits in it (see

[ZW04] for more details).

Entity. We extract *fine-grained entities* from source files. We distinguish the following major types of entities: file, class, method, field, and structure. Every entity is associated a name and a file which it is a part of.

Range. Entities are attributed to source files. We use *line ranges* to store the precise location of every entity in each *revision* of the file. Additionally, every check-in is assigned a set of *added*, *modified* and *deleted* entities.

2.3 Bug Tracking Systems

A *bug tracking system* is a database of problems reported by users. Every problem report contains information about the failure, status of the bug and many additional information that help developers address the issue. The most popular bug tracking systems are BUGZILLA, MANTIS, GNATS and TRAC.

To mirror a BUGZILLA database, we use its XML export feature. Additionally, we import attachments and activities directly from the web interface of BUGZILLA. As a result we get the right pane from Figure 2.2, which contains (among others) the following information about every bug (the entire database is very similar to the original BUGZILLA database scheme described in [Bar04]):

Bug ID. Every bug is assigned a *unique identifier*. It helps developers track the status of the bug and is used to identify duplicates. The bug ID is very often included in the log message when the bug gets fixed.

Short Description. A user, reporting the bug, fills in a *short description* of the bug which summarizes the failure she has observed.

Status. When a user reports a bug, it is first marked as *new*. As soon as one of the developers reproduces the failure, the status of the bug is changed to *verified* and often immediately gets *assigned* to one of the team members and eventually turns into *closed*. If the solution is not satisfying, the reporter may *reopen* the bug.

Resolution. When the bug gets *closed*, it is assigned a *resolution*, which may be one of: *duplicate*, *fixed*, *invalid*, *later*, *remind*, *won't fix* or *works for me*. Our database not only contains the current resolution

of the bug, but also every modification of this value with the date when it was made.

Assigned To. Once a bug has been verified as a true problem, it gets assigned to one of the developers who is then in charge to address the issue. This person usually introduces necessary changes to the sources, commits them to the repository and closes the bug.

Attachments. Users may attach files to bug reports. They usually contain data necessary to reproduce the failure, but sometimes users attach fixes.

2.4 Summary

This section sketched the general idea of our approach and described how we import the version control system and the bug tracking system to a common database. In the next chapter we give a detailed description of linking transactions with bugs, which corresponds to the upper box between the panes from Figure 2.2.

Chapter 3

Linking Transactions to Bugs

In order to locate fix-inducing changes, we first need to know if a change is a fix. A common practice among developers is to include a *bug report number* in the comment whenever they fix a defect associated with it. Čubranić and Murphy [ČM03] as well as Fischer, Pinzger, and Gall [FPG03a, FPG03b] exploited this practice to link changes with bugs. Figure 3.1 sketches the basic idea of this approach.

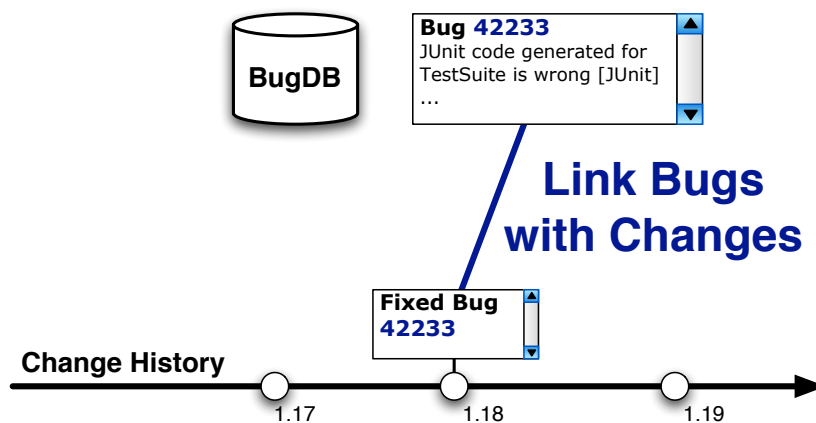


Figure 3.1: Link transactions to bug reports

In our work, we refine these techniques by assigning every link (t, b) between a transaction t and a bug b two independent levels of confidence: a *syntactic* level, inferring links from a CVS log to a bug report, and a *semantic* level, validating a link via the bug report data. These levels are later used to decide which links shall be taken into account in our experiments.

3.1 Syntactic Analysis

In order to find links to the bug database, we split every log message into a stream of tokens. A token is one of the following items:

- a *bug number*, if it matches one of the following regular expressions (given in FLEX syntax):
 - `bug[# \t]*[0-9]+`,
 - `pr[# \t]*[0-9]+`,
 - `show_bug\.cgi\?id=[0-9]+`, or
 - `\[[0-9]+\]`
- a *plain number*, if it is a string of digits `[0-9]+`
- a *keyword*, if it matches the following regular expression:
`fix(e[ds])?|bugs?|defects?|patch`
- a *word*, if it is a string of alphanumeric characters

Every number is a potential link to a bug. For each link, we initially assign a syntactic confidence *syn* of zero and raise the confidence by one for each of the following conditions that is met:

1. The number is a *bug number*.
2. The log message contains a *keyword*,
or the log message contains only *plain* or *bug numbers*.

Thus the syntactic confidence *syn* is always an integer number between 0 and 2. As an example, consider the following log messages:

- **Fixed bug 53784: .class file missing from jar file export**
The link to the bug number 53784 gets a syntactic confidence of 2 because it matches the regular expression for **bug** and contains the keyword **fixed**.
- **52264, 51529**
The links to bugs 52264 and 51529 have syntactic confidence 1 because the log message contains only numbers.
- **Updated copyrights to 2004**
The link to the bug number 2004 has a syntactic confidence of 0 because there is no syntactic evidence that this number refers to a bug.

3.2 Semantic Analysis

In the previous section, we inferred links that point from a transaction to a bug report. To validate a link (t, b) we take information about its transaction t and check it against information about its bug report b . Based on the outcome we assign the link a semantic level of confidence.

Initially, a link (t, b) has semantic confidence of 0 which is raised by 1 whenever one of the following conditions is met:

- The bug b has been resolved as **FIXED** at least once.¹
- The short description of the bug report b is contained in the log message of the transaction t .
- The author of the transaction t has been assigned to the bug b .²
- One or more of the files affected by the transaction t have been attached to the bug b .

This list is not meant to be exhaustive. One could for example check whether a change has been committed to the repository within a small time-frame around the time when a bug has been closed.³

Consider the following examples from ECLIPSE, which all have low confidence levels:

- **Updated copyrights to 2004**
The potential bug report number “2004” is marked as *invalid* and thus the semantic confidence of the link is zero.
- **Fixed bug mentioned in bug 64129, comment 6**
The number “6” appears in the comment for a fix. The syntactic confidence is 1, but the semantic confidence is 0.
- **Support expression like (i)+= 3; and new int[] {1}[0]
+ syntax error improvement**
“1” and “3” are (mistakenly) interpreted as bug report numbers here. Since the bug reports 1 and 3 have been fixed, the links both get a semantic confidence of 1.

¹Notice that only 27% of all bugs in the MOZILLA project are **FIXED** (47% for ECLIPSE).

²For this check, we need a mapping between the CVS and BUGZILLA *user accounts* of a project. For ECLIPSE, we mapped the accounts of the most active developers manually; for MOZILLA, we derived a simple heuristic based on the observation that email addresses were used as logins for both CVS and BUGZILLA.

³Čubranić and Murphy already applied this as a stand-alone technique to relate bugs to transactions in their HIPIKAT tool [ČM03].

- **Fixed bug 53784: .class file missing from jar file export.** The bug 53784 has not been closed, but resolved as LATER. Its short description is: “Different results when running under debugger” and author of the change has not been assigned this bug. Thus the semantic confidence of the link is 0.

However, there exists a bug 53284 with the following short description: “.class file missing from jar file export”. If the comment had contained a correct number, the link would be assigned the semantic confidence 3.

3.3 Quality

We have identified 25,317 links for ECLIPSE, connecting 47% of fixed bugs with 29% of transactions, and 53,574 links for MOZILLA, connecting 55.30% of fixed bugs with 43.91% of transactions.

Basing on a manual inspection of several randomly chosen links (see Section 3.2 for some examples), we have decided to use only those links whose syntactic and semantic levels of confidence satisfy the following condition:

$$sem > 1 \vee (sem = 1 \wedge syn > 0) \quad (3.1)$$

The intuition behind it is very simple:

1. If the *semantic level* of the link equals 0, there exists absolutely *no* semantic evidence that the transaction t brought a fix for the bug b .
2. If the *semantic level* of the link equals 1 and *syntactic level* equals 0, there exists a weak semantic evidence, but no syntactic one.

Tables 3.1 and 3.2 summarize the distribution of links across different classes of syntactic and semantic levels for both projects. It is worth noticing that fractions of links in each *sem/syn class* are very similar in both projects, which is probably due to the fact that their developers share similar practices. Our criterion discriminates less than 10% of links for both projects, which gives us a fairly large base of links that may be used in further experiments.

In order to check the quality of linking algorithm we have drawn 100 random links and checked manually whether they are correct (i.e. whether they are fixes of the bug they point to) or not. The breakdown of the inspected links across *sem/syn classes* is presented in Table 3.3. Every cell contains the number of links manually classified as correct and incorrect. As an example, consider the cell labeled with semantic value 2 and syntactic value 1. Out of 41 links that fall into this category, 39 were manually classified as

semantic	syntactic			total
	0	1	2	
0	270 (1%)	324 (1%)	110 (0%)	704 (3%)
1	1,287 (5%)	4,152 (16%)	1,922 (8%)	7,361 (29%)
2	2,057 (8%)	9,265 (37%)	2,421 (10%)	13,743 (54%)
3	1,439 (6%)	1,581 (6%)	482 (2%)	3,502 (14%)
4	2 (0%)	5 (0%)	0 (0%)	7 (0%)
total	5,055 (20%)	15,327 (61%)	4,935 (19%)	25,317 (100%)

Table 3.1: Distribution of links across different classes of syntactic and semantic confidence levels in ECLIPSE

semantic	syntactic			total
	0	1	2	
0	560 (1%)	1,211 (2%)	478 (1%)	2,249 (4%)
1	2,899 (5%)	9,059 (17%)	5,250 (10%)	17,208 (32%)
2	4,281 (8%)	16,336 (30%)	9,133 (17%)	29,750 (55%)
3	639 (1%)	2,241 (4%)	1,645 (3%)	4,525 (8%)
4	8 (0%)	22 (0%)	12 (0%)	42 (0%)
total	8,387 (16%)	28,669 (54%)	16,518 (31%)	53,574 (100%)

Table 3.2: Distribution of links across different classes of syntactic and semantic confidence levels in MOZILLA

semantic	syntactic			total
	0	1	2	
0	0/0	0/2	0/0	0/2
1	1/6	15/0	9/0	25/6
2	5/3	39/2	7/0	51/5
3	5/0	6/0	0/0	11/0
4	0/0	0/0	0/0	0/0
total	11/9	60/4	16/0	87/13

Table 3.3: Manual classification of links in ECLIPSE as correct / incorrect.

correct and the remaining 2—as incorrect. According to our discrimination criterion, these links are all classified as valid.

The overall comparison of manual and automatic classification is presented in Table 3.4. 94% of the time, both the manual and automatic classification put changes in the same class.

It is easy to notice that there exists no other discrimination criterion (based on link’s confidence levels) that would let us improve the accuracy of classification (since we reject only those classes where number of incorrect links is greater than the respective number of correct links).

3.4 Summary

We have presented an extension of the well-known algorithm for linking transactions with bugs. All the extracted links that pass the discrimination criterion (3.1) are later used for finding *fix-inducing changes*—the topic of the next chapter.

automatic	manual		total
	correct	incorrect	
valid link	86	5	91
invalid link	1	8	9
total	87	13	100

Table 3.4: Comparison of automatic and manual classification

Chapter 4

Locating Fix-Inducing Changes

Having isolated fixes, we determine *the earlier changes* which caused these fixes. We call such changes *fix-inducing*.

Definition 4.1 (fix-inducing change) *A change which is later undone by a fix is called fix-inducing unless it was introduced after the fixed problem had been reported.*

Notice that a change is not known to be fix-inducing until the corresponding fix is done.

4.1 Example

Consider the example in Figure 4.1, where revision 1.18 has been identified as a fix. We first compute the differences between the earlier revision (1.17) and the fixed revision (1.18). As a result we get the lines that have been changed by the fix.

Next, we annotate each line of the earlier revision (1.17) with the most recent author and revision that touched this line. Figure 4.2 shows the output of such an annotation. Right now, we use the CVS *annotate* command, but it is straightforward to implement a similar feature for other version control systems.

These annotations and the set of changed lines give us a set of *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, we use the bug database to rule out changes that cannot be fix-inducing because they were made after the bug had been reported, and thus could not be real causes for the bug. In our example, revisions 1.14

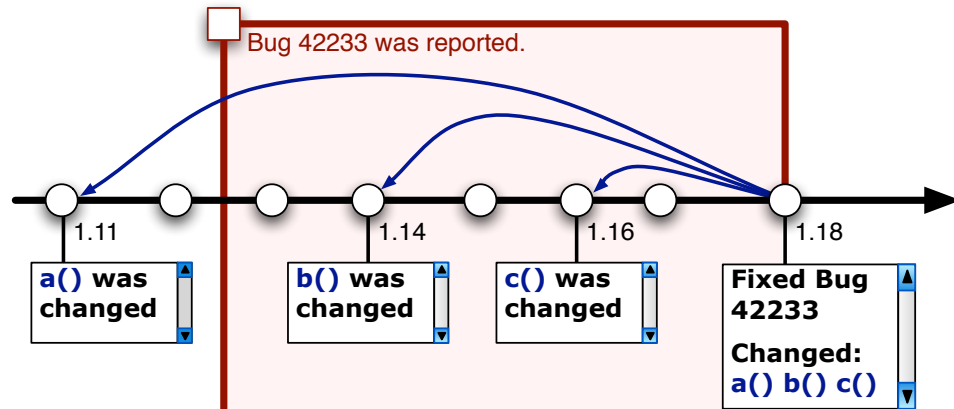


Figure 4.1: Locating fix-inducing changes for bug 42233

```

$ cvs annotate -r 1.17 Foo.java
...
19: 1.11 (john 12-Feb-03): public int a() {
20: 1.11 (john 12-Feb-03):     return i/0;
...
39: 1.10 (mary 12-Jan-03): public int b() {
40: 1.14 (kate 23-May-03):     return 42;
...
59: 1.10 (mary 17-Jan-03): public void c(){
60: 1.16 (mary 10-Jun-03):     int i=0;
...

```

Figure 4.2: CVS annotations for Foo.java

Input: Relation J , Relation L
Output: Set S

```

 $S \leftarrow \emptyset$ 
for each  $(b, c_f)$  in  $L$ 
  for each  $(c_d, c_f)$  in  $J$ 
    if  $timestamp(c_d) < timestamp(b)$  then
       $S \leftarrow S \cup \{c_d\}$ 
    end if
  end for
end for

```

Table 4.1: Algorithm for finding fix-inducing changes

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.

4.2 Formal Description

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:

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

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. The algorithm for detecting fix-inducing changes is summarized in Table 4.1.

	fix-inducing	\neg fix-inducing	all
fix	3.82 \pm 26.32	2.08 \pm 7.42	2.73 \pm 7.87
\neg fix	11.30 \pm 63.02	2.77 \pm 14.94	3.81 \pm 26.32
all	7.49 \pm 44.37	2.61 \pm 13.66	3.52 \pm 22.81

Table 4.2: Average sizes of fix and fix-inducing transactions for ECLIPSE

4.3 Results

Two results concerning fix-inducing changes are presented below:

Fix-Inducing Transactions are Large. In the first experiment we show that transactions that contain fix-inducing changes touch three times more files on average than their complement.

Do not Program on Fridays. We have created a breakdown of changes to days of week and found that fix-inducing changes are more likely to be introduced on Friday.

Both results seem to be intuitive, thus providing us evidences that our algorithm works well.

4.3.1 Fix-Inducing Transactions are Large

In our first experiment, we examined whether the span of the transaction (i.e. the number of files touched) correlates with the fact that the transaction is fix-inducing. Table 4.2 presents the average sizes of transactions for ECLIPSE. The transactions are split into four classes, depending on whether the transaction is a fix, fix-inducing, both, or none. For instance, the top-left cell means that the average size of transactions which are fixes *and* induce later on a fix is 3.82 (with a standard deviation “ \pm ” of 26.32).

Additionally, Table 4.2 shows that fix-inducing transactions are roughly three times larger than non fix-inducing transactions. Table 4.3 presents the same breakdown for MOZILLA which shows a similar trend.

Such data can be automatically retrieved from all projects that supply both a version archive and a bug database. It is especially worthy when deciding where to spend efforts in *quality assurance*. If we were in charge of the ECLIPSE project, for instance, we would take care that large extensions are well reviewed and tested, as these have a high potential for inducing later fixes.

	fix-inducing	¬fix-inducing	all
fix	5.79± 37.37	2.12± 9.74	4.39±30.05
¬ fix	4.61± 30.59	1.91±10.30	3.05±21.39
all	5.19± 34.12	1.97±10.13	3.58±25.23

Table 4.3: Average sizes of fix and fix-inducing transactions for MOZILLA

4.3.2 Do not Program on Fridays

We broke down changes by the day of the week when they were applied. We distinguished between *bugs* as indicated by fix-inducing changes, and *fixes* as detected by links to the bug database. Bugs may be also fixes, we refer to such changes as *fix-inducing fixes*; a similar concept of fix-on-fix have been previously used for visualization by Baker and Eick [BE94]. Finally, there are changes that are no bugs and no fixes.

$$P(\text{fix}) + P(\text{bug}) - P(\text{bug} \cap \text{fix}) + P(\neg \text{bug} \cap \neg \text{fix}) = 100\%$$

We measured the frequencies of the categories mentioned above. Table 4.4 presents the results for ECLIPSE. The likelihood $P(\text{bug})$ that a change will induce a fix is highest on Friday. The same holds for MOZILLA (see Table 4.5). Friday is the day where most ECLIPSE developers do fixes, for MOZILLA this is Sunday.

We used fix-inducing fixes to investigate whether non-fixes or fixes are more likely to be fix-inducing. Table 4.4 shows that for ECLIPSE, the average likelihood of introducing a fix-inducing change is almost three times higher for fixes, indicated by $P(\text{bug} | \text{fix})$, than for regular changes, indicated by $P(\text{bug} | \neg \text{fix})$. This does not hold for MOZILLA (see Table 4.5). The risk that a fix will be later undone is highest for ECLIPSE on Saturdays, and for MOZILLA on Fridays.

Almost every second change in MOZILLA is a fix and two out of five changes are fix-inducing. In the future we will investigate MOZILLA to find out what makes MOZILLA risky.

Besides the day of week, one can easily determine further properties of a change that correlate with inducing fixes—such as the development group, or the involved modules. Again, all this data is automatically retrieved for arbitrary projects.

4.4 Summary

As soon as a project has a bug database and a version archive, we can link the two to identify those changes that caused problems. Such fix-inducing changes have a wide range of applications. In this chapter, we examined the properties of fix-inducing changes in the ECLIPSE and MOZILLA projects and found, among others, that the larger a change, the more likely it is to induce a fix; checking for other correlated properties is straight-forward. We also found that in the ECLIPSE project, fixes are three times as likely to induce a later change than ordinary enhancements. Such findings can be generated automatically for arbitrary projects. In the next chapter we are examining one more application of fix-inducing changes—possibility of finding the most risky entities in the archive.

in %	Day of Week							avg
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
$P(\text{fix})$	18.4	20.9	20.0	22.3	24.0	14.7	16.9	20.8
$P(\text{bug})$	11.3	10.4	11.1	12.1	12.2	11.7	11.6	11.4
$P(\text{bug} \cap \text{fix})$	4.6	4.8	4.6	5.2	5.6	4.5	4.5	4.9
$P(\text{bug} \text{fix})$	25.1	22.9	23.3	23.5	23.2	30.3	26.4	23.7
$P(\text{bug} \neg \text{fix})$	8.2	7.1	8.1	8.8	8.7	8.4	8.6	8.1

Table 4.4: Distribution of fixes and fix-inducing changes across day of week in ECLIPSE

in %	Day of Week							avg
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
$P(\text{fix})$	42.5	46.5	49.7	45.9	48.4	50.2	61.1	48.5
$P(\text{bug})$	39.1	44.1	41.2	40.8	46.2	44.9	26.4	41.5
$P(\text{bug} \cap \text{fix})$	19.4	23.6	22.8	21.6	26.9	19.6	13.2	21.9
$P(\text{bug} \text{fix})$	45.7	50.8	45.8	47.1	55.6	39.1	21.6	45.2
$P(\text{bug} \neg \text{fix})$	34.1	38.3	36.7	35.5	37.3	50.6	33.9	38.1

Table 4.5: Distribution of fixes and fix-inducing changes across day of week in MOZILLA

Chapter 5

Evaluation

Let us once more remind the general idea of our approach. We start by *linking* bugs from a bug tracking system to changes made to the source and get a precise location of the fix. Next, we extract the changes that have *induced* those fixes, i.e. have introduced the fixed code. We have already seen that transactions that span across many files are more likely to be fix-inducing. What other conclusions can we draw about fix-inducing changes?

Conjecture 5.1 *Entities that had many fix-inducing changes in the past are more difficult to change than the others and thus are more risky to change.*

To verify this hypothesis we need two things: a definition of risk and an experiment that would support (or refute) the claim.

Definition 5.2 (risk) *We define a notion of risk as a percentage of fix-inducing changes in the history of the entity e ¹.*

$$risk(e) = \frac{\left| \left\{ r \mid r \text{ is a fix-inducing change that affects } e \right\} \right|}{\left| \left\{ r \mid r \text{ is a change that affects } e \right\} \right|}$$

Given a point of time t , we can split the history of an entity e into two parts: *past* and *future*. This way we obtain the definition of the *future risk*.

Definition 5.3 (future risk) *We define a notion of future risk as a percentage of fix-inducing changes introduced to the entity e after the point of time t .*

$$f_t(e) = \frac{\left| \left\{ r \mid r \text{ is a fix-inducing change that affects } e \text{ after } t \right\} \right|}{\left| \left\{ r \mid r \text{ is a change that affects } e \text{ after } t \right\} \right|}$$

¹ See section 5.3 for a more formal set of definitions.

non-relative predictor	relative to	referenced in
size of file	n/a	[OW02], [OWB04], [FO99], [FN99], [GKMS00]
lines added	size of file sum of sizes	[NB05]
# of changes	n/a	[GKMS00], [HH05]
# of fixes	# of changes	[MW00], [HH05], [BE94]
# of fix-inducing changes	# of changes	[SZZ05b]
# of fix-inducing fixes	# of changes	[SZZ05b]

Table 5.1: List of predictors used in the experiment

The risk of future changes used to be defined with the number of fixes in the future or after the release (see [OW02], [FO99], [GKMS00], [NB05]). This was simply because the researchers wanted to predict how many defects there were in the code. In contrast to that approach, we are not interested in locating the code that contains the highest number of defects, but in locating the code which is most risky to change, that is the code where a change is likely to introduce a defect. For this reason we use a fraction of fix-inducing changes instead.

The remainder of this chapter is organized as follows: Section 5.1 describes the settings we have used for our evaluation, whereas Section 5.2 describes the data we have used for our experiment. The details are presented in Section 5.3 followed by the results in Sections 5.4 and 5.5. Finally, Section 5.6 closes with discussion.

5.1 Evaluation Setup

We have implemented several predictors that use different measures to predict the risk of the change. Every predictor is able to give a prediction at any point of time. All the predictors are also briefly summarized in Table 5.1.

Size of the file. Size of the file is one of the most frequently referenced measures in defect prediction studies. It has been used as a sole predictor as well as a part of many statistical models (e.g.: [OW02], [OWB04], [FO99], [FN99], [GKMS00]). It is generally believed that size of the file has an important impact on the risk of the change, although type and direction of dependence varies between publications.

Our implementation counts the number of line terminators in a file. We use this predictor for text files, but have no implementation for finer entities.

Number of lines added. This predictor is based on recent work by Nagappan and Ball [NB05]. In our experiments we use three flavors of this measure:

Non-relative. This measure is the sum of number of lines added to the file since it has been imported to the repository; it is exactly the same value as *churned LOC* in the aforementioned work [NB05]².

Relative to sum of sizes. This is the non-relative value divided by the sum of the sizes of the file until the point of time when prediction is made.

Relative to size. This is the non-relative value divided by the size of the file at the time when prediction is made.

Number of changes. One of the common beliefs is that files with large number of revisions are more risky to change than those with shorter history. Graves et al. showed that the number of modifications to a file is a good predictor of the fault potential [GKMS00]. Hassan and Holt [HH05] use the number of changes as a heuristic that helps finding most risky files with the best *hit rate* among all the heuristic they tested.

Our implementation counts revisions of the file until the point of time when prediction is made.

Number of fixes. Fixing is generally considered more difficult than adding new features [MW00] and number of fixes applied to a file is one of the most popular measures used to express how bug-prone the file was in the past [HH05] and to visualize the distribution of faults among files [BE94].

Chapter 3 describes how we decide whether a change is a fix or not. Those changes that introduced an entity are not taken into account (because they do not fix this entity).

We use two flavors of this measure: **non-relative** and **relative** to the number of changes so far.

² A careful reader may notice that CVS counts only numbers of added and deleted lines, however since it does not distinguish modified lines at all, both numbers of added and deleted lines include the number of modified lines (a modification consists of deletion of the old line and addition of the new one).

Number of fix-inducing changes. We believe that the number of fix-inducing changes is one of the most important factors in predicting the future risk. This measure is our main contribution and, to our knowledge, it has not been studied yet.

Chapter 4 describes how we decide whether a change is fix-inducing or not. Our implementation ensures that the predictor is unaware that a change is fix-inducing unless the corresponding fix has already been introduced.

We use two flavors of this measure: **non-relative** and **relative** to the number of changes so far.

Number of fix-inducing fixes. This measure is only a simple modification of the previous one in which we count only those fix-inducing changes that are fixes as well.

It is available as both **non-relative** and **relative**.

5.2 Data Collection

We have linked transactions to fixes and extracted fix-inducing changes for three open source projects:

Eclipse “is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools.”³

Mozilla “is an open source project that was founded in order to create the next-generation Internet suite for Netscape. Mozilla Foundation now creates and maintains the Mozilla Firefox browser and Mozilla Thunderbird e-mail application, among other products.”⁴

Columba “is an e-mail client written in Java, featuring a user-friendly graphical interface with wizards and internationalization support.”⁵

We have chosen these projects because they are all mature, use CVS as their repository and give easy access to it. Their developers frequently insert bug numbers in commit messages, thus giving us a large quantity of data for analysis.

³Quotation from <http://www.eclipse.org> (August 1, 2005)

⁴Quotation from <http://en.wikipedia.org> (August 1, 2005)

⁵Quotation from <http://columba.sourceforge.net> (August 1, 2005)

	ECLIPSE	MOZILLA	COLUMBA
Language	Java	C/C++	Java
Start	28-04-2001	28-03-1998	08-04-2001
End	20-01-2005	20-01-2005	06-07-2005
Transactions	78,954	19,658	2,848
Revisions	278,010	392,972	18,109

Table 5.2: Summary of projects used in the experiment

Our database contains not only information about individual files and their history, but also histories of fine-grained entities: classes, methods, structures, fields, and so on. We have conducted our experiments in two setups:

Mixed granularity. A versatile predictor should be able to give predictions for fine-grained entities in source files but it should also fall-back to file-granularity level in case of non-source files. In this setup predictors are asked to estimate a risk of change to functions, structures and global variables in case of source files, and a risk of change in the file in case of text files.

Coarse granularity. Several predictors (e.g., size of the file) are only able to give an estimate of the risk for files. In this setup predictors are only asked about the risk of the change in the file, no matter if it is a source file or not.

In our experiments we disregard binary files (images, sounds and other data files). This is due to the fact that our method of locating fix-inducing changes works for text files only. Locating fix-inducing changes could be extended to binary files as well if only there existed a well defined semantic of *logical chunk* (like *line* in case of text files). CVS has no such abilities and new revision of binary file always replaces the old one as a whole.

5.3 Experiment Description

Our experiment has been designed to compare the accuracies of different predictors in a situation that is likely to occur in the daily work of a developer. We consider the situation when a developer uses a predictor to browse through the list of most risky locations in her code. We would like to know whether these entities are going to be risky in the future indeed.

A general scheme of the experiment is shown in the Figure 5.1. We take an arbitrary point of time (e.g. April 1, 2003) and compute predictions for every entity that exists in this particular moment (notice that the function `baaz()` is also taken into account). These predictions imply a ranking of the entities. On the other hand, we can create a similar ranking according to the actual (future) risks of those entities.

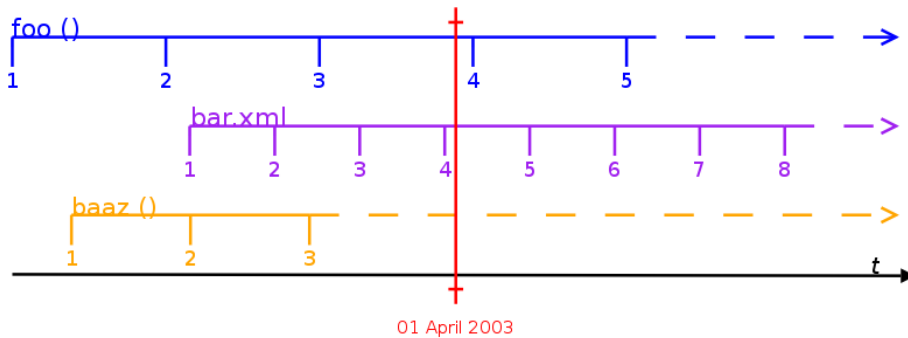


Figure 5.1: A general scheme of the experiment

We express the actual risk of changing an entity in the selected point of time as a fraction of fix-inducing changes after that point. If the entity has no revision past the selected point of time (like the function `baaz()` in our example), we set the probability of introducing a failure in the future to 0 as a consequence of simple rule: no change—no risk.

As pointed at the beginning, we want to know how many top-ranked entities are going to be most risky in the future. To check this, we use the well defined notion of *recall*, which is frequently referenced in *information retrieval* [BYBYRN99]:

$$recall = \frac{|result \cap wanted|}{|wanted|} \quad (5.1)$$

In the above definition *result* means the actual outcome of the prediction and *wanted*—the desired outcome. For a formal description we introduce the following definitions and notations:

- let E be a set of entities taken for evaluation,
- let $p : E \rightarrow \mathbb{R}_{\geq 0}$ be a predictor,
- let $after(e, t)$ be the set of revisions that affect the entity e after t ,

- let $fix_ind(S)$ be defined as $\{r \in S \mid r \text{ is fix-inducing}\}$,
- let r_p be an array of entities such that

$$\forall i \geq 0. p(r_p[i]) \geq p(r_p[i+1])$$

An array r_p is a ranking: the element $r_p[0]$ is the highest-ranked entity according to the predictor p , $r_p[1]$ is the second highest-ranked entity, and so on. $r_p[i \dots j]$ is a shortcut notation for $\{e \mid e = r_p[k] \wedge i \leq k \leq j\}$

- let $f_t : E \rightarrow \mathbb{R}_{\geq 0}$ be defined as:

$$f_t(e) = \begin{cases} \frac{|fix_ind(after(e, t))|}{|after(e, t)|} & \text{if } after(e, t) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

f_t is the future risk function informally introduced in Definition 5.3. The second part of the formula is a consequence of the rule *no change—no risk*.

We are interested in the recall of the *most risky* entities in the set of top- n entities ranked by a predictor p . By *most risky* we mean the set of entities that have a future risk greater than some $\varepsilon > 0$. We replace *result* and *wanted* sets in the general formula for *recall* (5.1) with the following sets:

$$\begin{aligned} result &= \{e \mid e \in r_p[0 \dots n]\} \\ wanted &= \{e \mid f_t(e) > \varepsilon\} \end{aligned}$$

After parameterization we get the following formula:

$$recall_{\varepsilon, p, t}(n) = \frac{|\{e \mid f_t(e) > \varepsilon \wedge e \in r_p[0 \dots n]\}|}{|\{e \mid f_t(e) > \varepsilon\}|}$$

We do not inspect the complementary notion of *precision*:

$$precision = \frac{|result \cap wanted|}{|result|} \quad (5.2)$$

This is due to the fact that its plot is more difficult to analyze (in contrast to a monotone function of *recall*) and gives the same result in case $|result| = |wanted|$.

$$precision = \frac{|result \cap wanted|}{|result|} = \frac{|result \cap wanted|}{|wanted|} = recall$$

5.4 Results

Before we present the results, we would like to explain why we separate relative and non-relative predictors in the plots. It is easy to notice that non-relative predictors prefer older entities (large number of fixes implies large number of changes), whereas relative predictors are less sensitive to the length of the history.

We have chosen 12 points of time: first days of months from April 1, 2003 until March 1, 2004. We have computed the *recall* for each of them and took an average. The computations have been performed with $\varepsilon = 0.3$. Fraction of files with future risk greater than this value varies in the range 5–6% for all the three projects (1313 out of 15017 for ECLIPSE, 2121 out of 31292 for MOZILLA and 21 out of 398 for COLUMBA on April 1, 2003). Figures 5.2 and 5.3 present the results of the experiment for ECLIPSE, Figures 5.4 and 5.5 present analogous results for MOZILLA and 5.6 and 5.7 for COLUMBA. Dashed lines indicate the argument value $n = |result| = |wanted|$.

For all the projects, all the predictors performed rather poor, achieving less than 10% recall. This suggests that simple measures are not accurate enough to return the most risky files. Although predictors based on fixes and fix-inducing changes have proven to be the best among the chosen ones, the difference between them and the rest is rather small in case of ECLIPSE and completely insignificant for the remaining projects.

We have also run our experiment on mixed-granularity level for April 1, 2003⁶. Table 5.3 summarizes the results for $recall_{\varepsilon,p,t}(|wanted|)$. They are even worse than file-granularity results—no predictor in any project has achieved *recall* of at least 2%.

⁶Results for COLUMBA have been computed for $\varepsilon = 0.2$ because there were no entity with future risk greater than 0.3.

predictor	ECLIPSE ($\varepsilon = 0.3$)	MOZILLA ($\varepsilon = 0.3$)	COLUMBA ($\varepsilon = 0.2$)
# of changes	0.000	0.000	0.000
# of fixes	0.011	0.000	0.000
# of fix-inducing changes	0.008	0.004	0.000
# of fix-inducing fixes	0.004	0.004	0.000
% of fixes	0.018	0.000	0.000
% of fix-inducing changes	0.008	0.000	0.000
% of fix-inducing fixes	0.009	0.000	0.000

Table 5.3: Mixed-granularity level for $recall_{\varepsilon,p,t}(|wanted|)$ results computed for April 1, 2003.

predictor	ECLIPSE	MOZILLA	COLUMBA
size of file	0.08	0.24	0.12
lines added	0.13	0.14	<u>0.33</u>
# of changes	<u>0.17</u>	0.28	<u>0.30</u>
# of fixes	0.15	<u>0.31</u>	0.21
# of fix-inducing changes	0.14	0.27	0.19
# of fix-inducing fixes	0.13	0.27	0.11
lines added / sum of sizes	0.06	0.03	0.03
lines added / size	0.04	0.07	0.04
% of fixes	0.14	0.25	0.09
% of fix-inducing changes	0.07	0.21	0.06
% of fix-inducing fixes	0.12	0.25	0.01

Table 5.4: Correlation coefficients for file-granularity level (April 1, 2003)

5.5 Correlation

In order to get a deeper insight into the data, we have also computed the linear correlation coefficients between values returned by predictors and future risk. The results are presented in Table 5.4. They show almost no correlation between future risk and predictions given by any of the predictors. Even worse: the highest correlation coefficients (underlined) are achieved by different predictors for all three projects. It may mean that different characteristics of these programs do not let give a universally best measure.

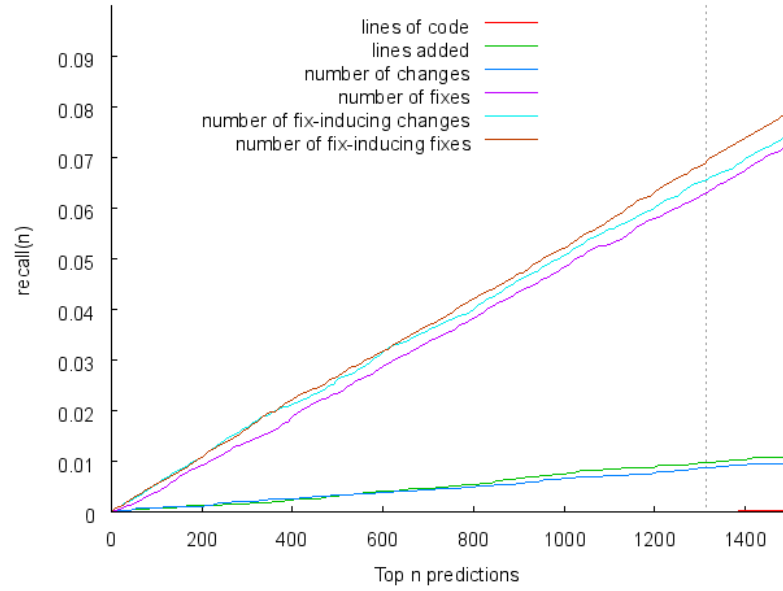


Figure 5.2: Experiment *recall* results for ECLIPSE, non-relative predictors, file-granularity level, $\varepsilon = 0.3$

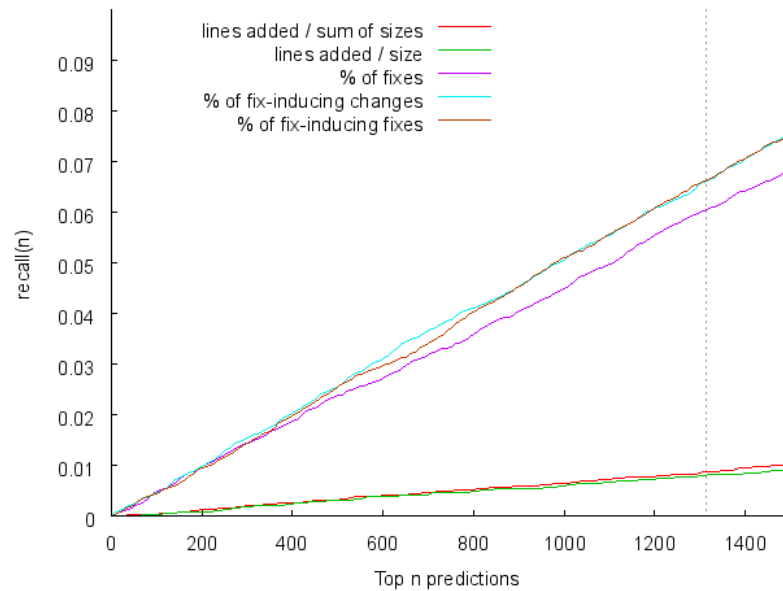


Figure 5.3: Experiment *recall* results for ECLIPSE, relative predictors, file-granularity level, $\varepsilon = 0.3$

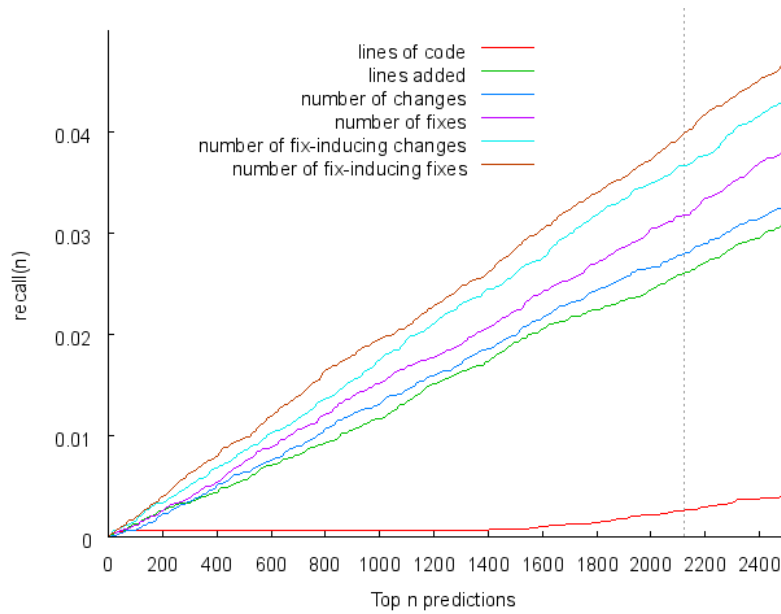


Figure 5.4: Experiment *recall* results for MOZILLA, non-relative predictors, file-granularity level, $\varepsilon = 0.3$

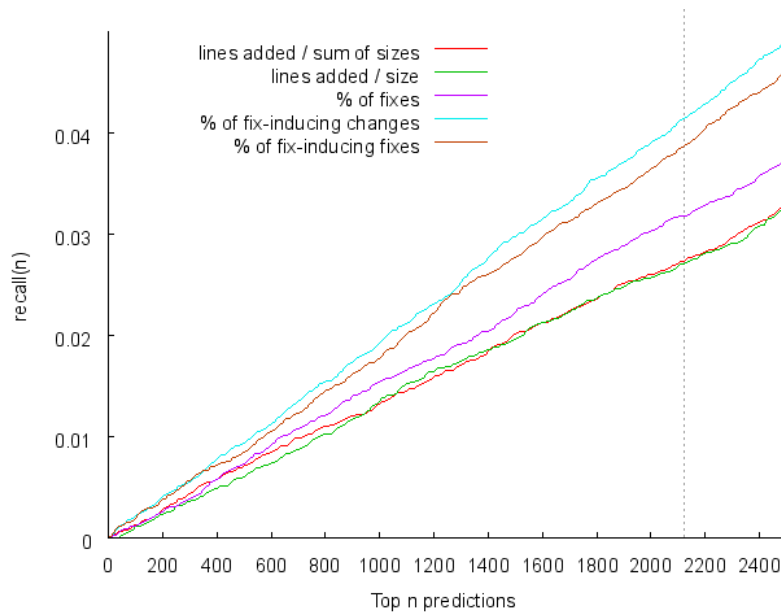


Figure 5.5: Experiment *recall* results for MOZILLA, relative predictors, file-granularity level, $\varepsilon = 0.3$

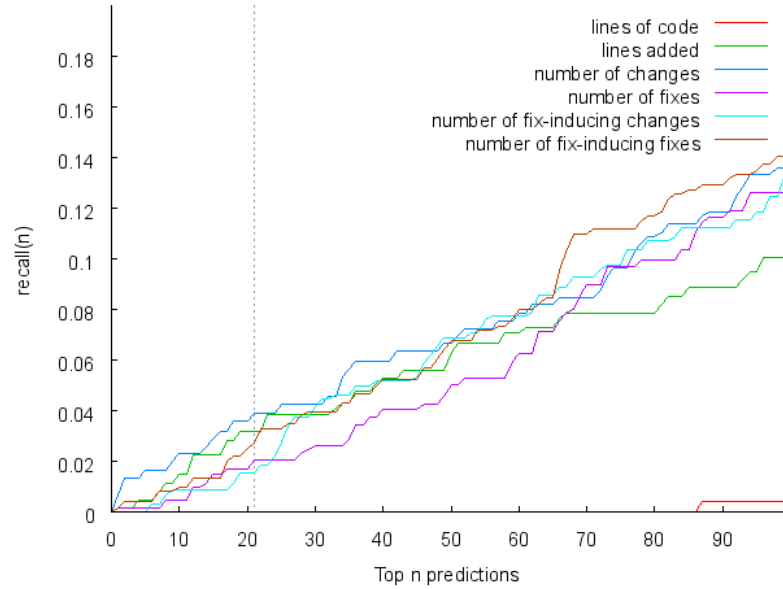


Figure 5.6: Experiment *recall* results for COLUMBA, non-relative predictors, file-granularity level, $\varepsilon = 0.3$

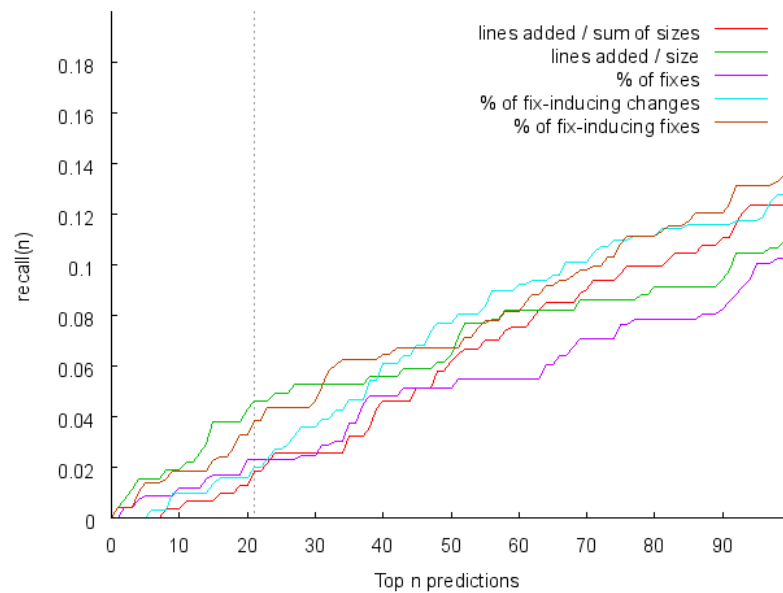


Figure 5.7: Experiment *recall* results for COLUMBA, relative predictors, file-granularity level, $\varepsilon = 0.3$

5.6 Discussion

There are many possible reasons why our experiments have failed. One obvious conclusion would be that our conjunction does not hold. However, there are several issues in the setup of the experiment that may have influenced the results. The ongoing research should investigate the following parameters of our experiments:

Sizes of past and future periods. The future risk of an entity with no revision past the chosen point of time has been defined to be 0. This is an arbitrary choice and it means in practice that the measures not only need to predict the risk, but also foresee whether an entity will be changed or not. It might be beneficial to focus on entities that have some minimal number of revisions past the selected point of time as well as on entities that have some minimal history.

Different file types. The experiment takes into account every text file in the repository. It could possibly let us improve the accuracy if we restricted ourselves to source files only.

Development stage. Correlation coefficients may be incomparable across projects because they are all at different stages of development. More points of time, not necessary parallel ones, could be investigated in order to draw more meaningful conclusions.

5.7 Summary

This chapter described the experiments we have used to evaluate the usability of our approach. It has turned out that it is extremely difficult to predict the risk of changing an entity and that no simple measure gives accurate results. However, there are several issues that still need to be investigated and they will be addressed in our future work. The next chapter describes work related to our approach.

Chapter 6

Related Work

In this chapter we describe the work that is directly related to our approach. We have split this chapter into two parts. In Section 6.1 we summarize the work in the area of mining software repositories and bug tracking systems, while Section 6.2 describes previous approaches to predicting defects in software systems.

6.1 Software Repositories and Bug Tracking Systems

One of the first papers on mining software repositories was a work by Ball, Kim, Porter and Siy [BKPS97] in which the authors present various techniques raising understanding of program's development history. Baker and Eick proposed a similar to fix-inducing changes concept of *fix-on-fix changes* [BE94]. Such changes focus on the induced fix, while we investigate the inducing change. Thus, every fix-on-fix change has a preceding fix-inducing change, but not every fix-inducing change results in a fix-on-fix.

In order to locate fix-inducing changes, we need first to *identify fixes* in the version archive. Mockus and Votta developed a technique that identifies reasons for changes (e.g. fixes) in the log message of a transaction [MV00]. In our approach, we refine the techniques of Čubranić and Murphy [ČM03] and of Fischer, Pinzger, and Gall [FPG03b, FPG03a], who identified references to bug databases in log messages and used these references to infer links from CVS archives to BUGZILLA databases.

Čubranić and Murphy additionally inferred links in the other direction that is: from BUGZILLA databases to CVS archives, by relating bug activities to changes. This has the advantage to identify fixes that are not referenced in log messages. For more details about this approach, we refer to [Čub04].

Rather than searching for fix-inducing changes, one can also directly determine *failure-inducing changes*, where the presence of the failure is determined by an automated test. This was explored by Zeller, applying Delta Debugging on multiple versions [Zel99].

6.2 Defect Prediction

Defect prediction is an area of software engineering that has been studied for many years now. Many approaches relate various metrics of code complexity in files with fault likelihood. Graves, Karr, Marron and Siy [GKMS00] as well as Ostrand, Weyuker and Bell [OWB04] argue that the risk is strongly associated with the size of the file. Fenton and Neil [FN99] as well as Fenton and Ohlsson [FO99] adopt the opposite stance.

Mockus and Weiss [MW00] were among the first ones to analyze the properties of the change instead of the properties of the changed file. Their tool estimates the probability that the change will introduce a defect, whereas our approach does not require any information about the change to be made because it focuses on the probability of introducing the defect sometime in the future, not necessarily in the next revision.

Nagappan and Ball [NB05] go even further and analyze the entire history of changes. They argue that the proportion of churned (added or modified) code to the size of the new binary is strongly correlated with the defect density. In contrast to this work, we measure the reliability of the software in terms of *fix-inducing changes*, both as a predictor of future risk and as a measure of true risk associated with a location.

Hassan and Holt [HH05] propose several heuristics for finding the most faulty files. More than 60% of the time one of the top 10 files sorted according to the number of fixes is fixed again. As in all former publications, they focus on *fixes*, whereas our approach makes use of the ability of blaming particular change as *fix-inducing*.

Chapter 7

Conclusions

As soon as a project has a bug database as well as a version archive, we can link the two to identify those changes that caused a problem. Such fix-inducing changes have a wide range of applications. In this paper, we examined the properties of fix-inducing changes in the ECLIPSE and MOZILLA projects and found, among others, that the larger a change, the more likely it is to induce a fix; checking for other correlated properties is straight-forward. We also found that in the ECLIPSE project, fixes are three times as likely to induce a later change than ordinary enhancements. Such findings can be generated automatically for arbitrary projects.

Next, we investigated whether several simple measures could allow us to predict the risk of future changes in software systems. Unfortunately, it turned out that the accuracy of pointing out the most risky entities in the repository is unsatisfactory. We also discussed several important issues that still need to be investigated, namely *sizes of past and future periods*, *different file types* and *development stage of the projects*.

Appendix A

Tools

Our approach consists of two phases: preprocessing and prediction (see Chapter 2 for details). We have created tools that perform computations for both phases. Section A.1 covers preprocessing tool and Section A.2—prediction tools.

A.1 Preprocessing

Importing CVS and BUGZILLA data to the database was not an objective of this thesis. Our tool assumes that these have already been mirrored to a common database. The `cvsbts` tool uses the following syntax:

```
cvsbts [OPTIONS] [COMMANDS]
```

Commands specify which part of the job should be done and it may be one (or more) of the following:

- `--link` Link CVS transactions with BUGZILLA bugs.
- `--fetch` Fetch annotations for fix revisions from the CVS repository.
- `--mark` Mark fix-inducing revisions.
- `--clean` Remove fix-inducing mark for those changes that may not be fix-inducing, because they have been made after the bug has been reported.

The following options affect the behavior of the tool:

- `--driver` Use the specified driver for connection with the database.
- `--server` Specify the database on which all operations are performed.

```
#!/bin/sh -e
# usage: revision file anndir cvsdir cvsroot

if test -f "$3/$2.$1"; then
    exit 0
fi

if test -d `dirname "$3/$2.$1"`; then
    install -d `dirname "$3/$2.$1"`
fi

cd $4
cvs -d$5 annotate -r "$1" "$2" > "$3/$2.$1"
```

Figure A.1: Example implementation of the `cvsann` tool

`--user` Use the specified username for connection with the database.

`--pass` Use the specified password for connection with the database.

`--cvsdir` Specify the directory with the project checked out from the repository. CVS `annotate` command is run in this directory with the `cvsann` tool (see below).

`--cvsroot` Specify the root of the CVS repository. This argument is passed to the `cvsann` tool (see below).

`--anndir` Specify the directory where annotations should be stored or where they are already stored.

The `cvsbts` tool may be used to locate fix-inducing changes for projects that use other version archive than CVS. It relies on the presence of the `cvsann` program (or script), which is used to fetch a single annotation from the repository. The `cvsann` tool is passed five arguments: a *revision number*, a *file name*, a *directory* where the annotation should be stored, a *directory* where the local copy of the sources is stored and a *location* of the CVS root. The script should enter the directory with the local copy of the sources, fetch the annotation and store it in the given directory. The script should return 0 if and only if it succeeded. An example implementation that we used with CVS is presented in Figure A.1. We have used the command presented in Figure A.2 to locate fixes and fix-inducing changes in ECLIPSE.

```
java org.softevo.cvsbts.Main \  
--server=jdbc:postgresql://localhost:5555/eclipse-cvs-bugs \  
--driver=org.postgresql.Driver \  
--user=slifers \  
--anndir=/scratch/zimmerth/Birne/ \  
--cvsdir=/scratch/zimmerth/Apfel/ \  
--cvsroot=:pserver:anonymous@dev.eclipse.org:/home/eclipse \  
--link \  
--fetch \  
--mark \  
--clean
```

Figure A.2: Example call of the `cvsbts` tool

A.2 Prediction

The evaluation consists of two steps. In the first step we collect the measures returned by the predictors and store them in a file. In the second phase we extract *recall* and *correlation* from this file.

A.2.1 Collecting Predictions

To collect the predictions we use the `Errasmus` tool with the following syntax:

```
Errasmus [OPTIONS]
```

The following options affect the behavior of the tool:

- `--driver` Use the specified driver for connection with the database.
- `--server` Specify the database on which all operations are performed.
- `--user` Use the specified username for connection with the database.
- `--pass` Use the specified password for connection with the database.
- `--user-evaluate` Specifies the comma-separated list of the dates, for which the predictions should be computed.
- `--out` Specifies the root of the filename in which the results are to be stored. Consecutive integer numbers (starting with 1) are appended to this name and predictions for consecutive dates are stored in these files.
- `--granularity-full` Run experiments on mixed granularity level instead of file granularity level (which is the default).

```

java -Xmx1024m org.softevo.errasmus.Errasmus \
--driver=org.postgresql.Driver \
--server=jdbc:postgresql://localhost:5555/eclipse-cvs-bugs \
--user=sliwers \
--user-evaluate="2003-04-01,2003-05-01,2003-06-01,\
2003-07-01,2003-08-01,2003-09-01,2003-10-01,2003-11-01,\
2003-12-01,2004-01-01,2004-02-01,2004-03-01" \
--repo-path=/scratch/zimmerth/CACHE/FILES \
--out=/scracth/sliwers/eclipse-user-eval

```

Figure A.3: Example call of the `Errasmus` tool

```

java -Xmx512m org.softevo.errasmus.Analysis \
--results=/scratch/sliwers/eclipse-user-eval7 \
--top-ten-recall

```

Figure A.4: Example call of the `Analysis` tool

`--repo-path` Path to the local copy of the repository is used to calculate the current size of each file.

Figure A.3 presents the command we have used to conduct our experiments on the database for ECLIPSE, on file granularity level. It has produced 12 files. The file `eclipse-user-eval7`, for example, contains predictions and future risks for October 1, 2003.

A.2.2 Analysis

In order to analyze the results and dump the important data we use the *Analysis* tool with the following syntax:

```
Analysis [OPTIONS] [COMMANDS]
```

Commands specify which part of the job should be done and it may be one (or more) of the following:

`--info` Dump a general information about the results.

`--top-ten-recall` Compute *recall* results.

`--values` Output future risk and predictions for all entities.

The following options affect the behavior of the tool:

`--results` Name of the file where results are stored.

Figure A.4 presents the command we have used to get the *recall* results for October 1, 2003 in our experiment.

Bibliography

- [Bar04] Nick Barnes. Bugzilla database schema. Technical report, Ravenbrook Limited, July 2004. <http://www.ravenbrook.com/project/p4dti/master/design/bugzilla-schema/>.
- [BE94] 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.
- [BKPS97] Thomas Ball, Jung-Min Kim, Adam A. Porter, and Harvey P. Siy. If your version control system could talk. . . . In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.
- [BYBYRN99] Ricardo A. Baeza-Yates, R. Baeza-Yates, and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [ČM03] Davor Čubranić and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, May 2003.
- [Čub04] Davor Čubranić. *Project History as a Group Memory: Learning From the Past*. PhD thesis, University of British Columbia, Canada, December 2004.
- [FN99] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(3), May/June 1999.

-
- [FO99] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 1999.
- [FPG03a] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, November 2003. IEEE.
- [FPG03b] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, September 2003. IEEE.
- [GKMS00] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7), 2000.
- [HAK⁺96] John P. Hudepohl, Stephen J. Aud, Taghi M. Khoshgoftaar, Edward B. Allen, and Jean Mayrand. Emerald: Software metrics and models on the desktop. *IEEE Software*, 13(5):56–60, 1996.
- [HH05] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the International Conference on Software Maintenance*, Budapest, Hungary, September 2005.
- [MV00] Audris Mockus and Lawrence 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, October 2000. IEEE.
- [MW00] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.
- [NB05] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering (ICSE)*, May 2005.

- [OW02] T. Ostrand and E.J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the ACM/International Symposium on Software Testing and Analysis (ISSTA2002)*, pages 55–64, Rome, Italy, July 2002.
- [OWB04] Thomas J. Ostrand, Elaine J. Weyuker, and Robert 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.
- [SZZ05a] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: Raising risk awareness. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Lisbon, Portugal, September 2005.
- [SZZ05b] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? On Fridays. In *Proc. International Workshop on Mining Software Repositories (MSR)*, Saint Louis, Missouri, USA, May 2005.
- [Zel99] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, volume LNCS 1687. Springer Verlag, 1999.
- [ZW04] Thomas Zimmermann and Peter 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.

Statement Under Oath

I confirm under oath that I have written the Thesis on my own and that I have not used any other media than the ones mentioned in the Thesis.

Saarbrücken, August 10, 2005.

Signature

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den. 10 August 2005.

Unterschrift