Who Knows about That Bug?

Automatic Bug Report Assignment with a Vocabulary-Based Developer Expertise Model

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Dominique Urs Philipp Matter

Juni 2009

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Adrian Kuhn

Institut für Informatik und angewandte Mathematik

Further information about this work and the tools used as well as an online version of this document can be found under the following addresses:

Dominique Urs Philipp Matter
matter@iam.unibe.ch
http://scg.unibe.ch/wiki/projects/develect

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Abstract

For popular software systems, the number of daily submitted bug reports is high. Triaging these incoming reports is a time consuming task. Part of the bug triage is the assignment of a report to a developer with the appropriate expertise. In this thesis, we present an approach to automatically suggest developers who have the appropriate expertise for handling a bug report. We model developer expertise using the vocabulary found in their source code contributions and compare this vocabulary to the vocabulary of bug reports. We evaluate our approach by comparing the suggested experts to the persons who eventually worked on the bug. Using eight years of Eclipse development as a case study, we achieve 33.6% top-1 precision and 71.0% top-10 recall. Validating these results with a case study on four years of Gnome/Evolution development, we achieve 19.2% top-1 precision and 64.7% top-10 recall.

Acknowledgements

The presented work was completed due to the support of many people. I would like to try to express my thanks, being aware of the fact that this enumeration might be incomplete in both its quantity and quality.

My supervisor Adrian Kuhn for his guidance, for providing—and motivating me with—fresh ideas and new input and investing a lot of time working with me.

Prof. Dr. Oscar Nierstrasz for letting me write my thesis in the Software Composition Group (SCG).

All the other members of the SCG and its affiliated people for their advice and contributions to this work.

All my friends for their support and for making this an enjoyable and entertaining time. Particularly Dominic Descombes for the numerous interesting and inspiring discussions and his input on this work.

Ann Katrin Hergert for providing me the fundamental motivation to keep working on and finally finish this work, as well as for proof reading the final draft of this thesis.

My family and especially my parents for their unconditional, never-ending support.

Thank you.

Contents

Chapter 1

Introduction

Software repositories of large projects are typically accompanied by a bug report tracking system. In the case of popular software systems, the bug tracking systems receive an increasing number of reports daily. The task of triaging the incoming reports therefore consumes an increasing amount of time [3]. One part of the triage is the assignment of a report to a developer with the appropriate expertise. Since in large software systems the developers typically are numerous and distributed, finding a developer with a specific expertise can be a difficult task [30].

Expertise models of developers can be used to support the assignment of developers to bug reports. It has been proposed that tasks such as bug triage can be improved if an externalized model of each programmer's expertise of the code base is available. The parts of the code for which the developer is an expert are indicated by the frequency and recency of interaction [15]. The best accessible kind of developer and code interactions are the developer's source code contributions. Even though approaches for expertise models based on software repository contributions are available, existing recommendation systems for bug assignment typically use expertise models based on previous bug reports only [4, 11, 40, 30, 14]. Typically a classifier is trained with previously assigned bug reports, and is then used to classify and assign new, incoming bug reports.

In this thesis, we propose an expertise model based on source code contributions. We apply in it a recommendation system that assigns developers to bug reports. We compare vocabulary found in the `diffs` of a developer's contributions with the vocabulary found in the description of a bug report. We then recommend developers whose contribution vocabulary is lexically similar to the vocabulary of the bug report.

We implemented our approach as a prototype called DEVELECT[1]. We evaluate the recommendation system using the Eclipse project and a project from Gnome as case

[1]DEVELECT is open source, written in Smalltalk, and available at `http://smallwiki.unibe.ch/develect`. The name *develect* is a portmanteau word of *developer* and *dialect*.

studies. We develop and calibrate our approach on a training set of bug reports from Eclipse. Then we report and discuss the results of evaluating it on a set of reports of the remaining Eclipse case study. We further use the project from Gnome as a case study to validate the previously obtained results.

Parts of this thesis have been previously published in the Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR 2009) [28]. The contributions of this previous work are as follows:

- We propose a novel expertise model of developers. The approach is based on the vocabulary found in the source code contributions of developers.
- We propose a recommendation system that applies the above expertise model to assign developers to bug reports. We evaluate the system using eight years of Eclipse development as a case study.
- We report on the decay of developer expertise, observed when calibrating our approach. We apply two weighting schemes to counter this effect.

The additional contributions of this thesis include the following:

- We provide further state of the art.
- We provide the detailed algorithm to build and apply our expertise model.
- We provide a more extensive discussion of the calibration of our approach and the threats to its validity.
- We validate the results obtained in the Eclipse case study using Gnome as an additional case study.
- We inspect the vocabularies used in bug reports and source code.

1.1 Our Approach in a Nutshell

Our approach covers both the construction of an expertise model of developers and the application of a recommendation system that uses this expertise model to automatically assign developers to bug reports. The approach requires a versioning repository to construct the expertise model as well as a bug report, possibly provided by a bug tracking facility, to assign developers to. We realized it as a prototype implementation called DEVELECT.

Our recommendation system is based on an expertise model of the developer's source code contributions. For each developer, we count the textual word frequencies in their change sets. This includes deleted code and context lines, assuming that any kind of change (even deletions) requires developer knowledge and thus familiarity with the vocabulary.

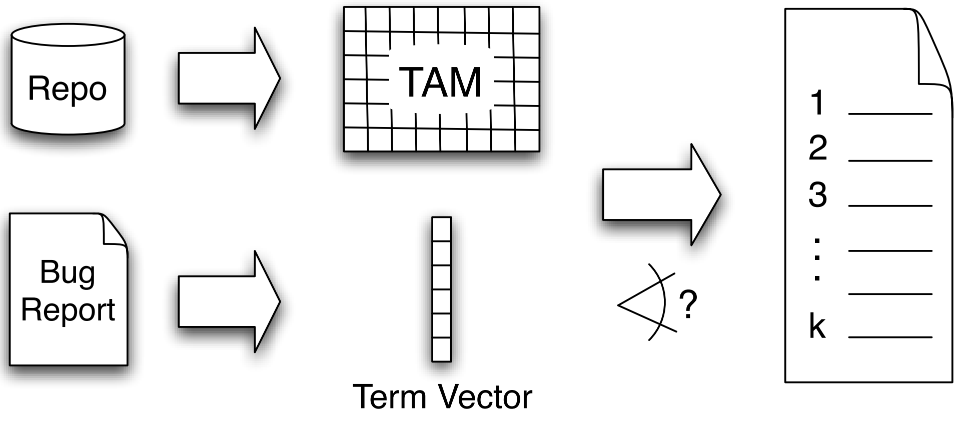

Figure 1.1: Architecture of the DEVELECT recommendation system: (left) bug reports and versioning repositories are processed to produce term vectors and term-author-matrices; (right) the cosine angle between vector and matrix columns is taken to rank developers in the suggested list of experts.

Given a software system with a versioning repository, the creation of the expertise model works as illustrated in Figure 1.1:

1. For each contributor to the versioning repository, we create an empty bag of words.
2. For each contribution to the versioning repository, we create a `diff` of all changed files and count the word frequencies in the diff files. The word frequencies are further weighted by a decay factor that is proportional to the age of the contribution. We assign the word frequencies to the contributor's bag of words.
3. We create a term-author-matrix $S_{n \times m}$, where n is the global number of words and m the number of contributors. Each entry $s_{i,j}$ equals the (weighted) frequency of the word t_i in the contributions of the contributor a_j.

Given the above term-author-matrix and a bug report, the assignment of developers works as follows:

1. We count the word frequencies in the bug report and create a query vector v of length n where v_i equals the frequency of the word t_i in the bug report.
2. For each developer in the term-author-matrix, we take the cosine of the angle between the query vector and the developer's column of the matrix. If the developer has been inactive for more than three months, this lexical similarity is decreased by a penalty proportional to the time since his/her latest contribution.

3. We rank all developers by their lexical similarity and suggest the top-k developers.

This list of suggested experts can be provided to a person performing bug triage. This person can use the list to find the developers with the appropriate expertise for handling a bug report. That is, developers who can be assignees of the report and/or help tracking and resolving the bug and its report, *e.g.* by commenting on the report and participate in its discussion. The list can also indicate if an already assigned report is appropriately assigned (in terms of expertise). However, the list does not state if the corresponding developers *e.g.* have time or are generally willing to handle it. Even if they do not, they might—due to their expertise—still point to other developers with similar expertise who could have time and be willing to assist.

To evaluate our approach we use Eclipse and Gnome as case studies. In each case, we train our system with weekly summaries of all contributions and use the resulting expertise model to assign developers to bug reports. We evaluate the precision and recall of our approach by comparing the suggested developers to the persons who eventually worked on the bug and its report.

For example, for a bug report that was submitted in May 2005, we would train our system with all commits up to April. Then we would evaluate the suggested developers against those persons who worked on the bug and its report in May or later on to see if they match.

1.2 Structure of the Thesis

In chapter 2 we discuss the current state of the art. In chapter 3 we provide the algorithm to build and apply our expertise model. In chapter 4 we evaluate our expertise model using Eclipse as a case study. In chapter 5 we discuss the calibration of our approach and threats to its validity. In chapter 6 we use a project from Gnome as a case study to validate the previously obtained results. In chapter 7 we briefly inspect the vocabularies used in bug reports and source code. In chapter 8 we conclude with remarks on future work.

Chapter 2

State of the Art

There exists previous work on mining developer expertise from source code. However, if such an expertise model is applied to assist in the triage of bug reports, it requires an explicit linkage between bug reports and source files [5]. Most existing recommendation systems for the assignment of developers to bug reports use other expertise models, as *e.g.* models based on previous reports. Thus, the main differences between our approach and other recommendation systems are: First, our system is trained on the source code repository rather than on previous bug reports. Second, for the recommendation of a developer suitable to handle a bug report, we do not need a linkage between the bug tracking system and the software repository. As a collorary, our approach is a) not dependent on the quality of previous bug reports, and b) can also be used to assign developers that have not worked on any bug report previously. The main requirement for a developer to be assignable is that he/she has a contribution history of approximately half a year or more.

2.1 Expertise Models

In this section, we present existing work related to expertise models and recommendation systems in general. In the next section, we present existing work including recommendation systems specific to bug triage.

In his work on automatic bug report assignment, John Anvik proposes eight types of information sources to be used by a recommendation system [3]. Our recommendation system focuses on three of these types of information: the information of the textual description of the bug (type 1), the developer who owns the associated code (type 6), and the information of the list of developers actively contributing to the project (type 8). We refine information type 6 to take into account the textual content of the code owned by a developer; our recommendation system is based on an expertise model of the developer's source code contributions. Our system currently does not consider the component the bug is reported for (type 2), the operation system that the bug occurs on (type 3), the hardware that the bug occurs on (type 4), the version of the software the bug was observed for (type 5), or the current workload of the developers (type 7). Information types 2–4 are indirectly covered, since textual references to the component, operation system, or hardware are taken into account when found in bug reports or source code. Information of type 7 is typically not publicly available for software projects and thus excluded from our studies. Furthermore, we deliberately disregard information of type 5, since developer knowledge acquired in any version pre-dating the bug report might be of use.

Mockus and Weiss [31] compute the experience of a developer as a function of the number of changes he/she has made to a software system so far. Additionally, they compute a recent experience by weighting recent changes more than older ones. Mockus and Herbsleb [30] then use such an experience to model the expertise of a developer. Furthermore, they examine the time that a developer needs to find additional people to work on a given modification request. Based on the results, they report that finding experts is a difficult task.

Fritz et al. [15] report on an empirical study that investigates whether a programmer's activity indicates knowledge of code. They found that the frequency and recency of interaction indicates the parts of the code for which the developer is an expert. They also report on a number of indicators that may improve the expertise model, such as authorship, role of elements, and the task being performed. In our work, we use the vocabulary of frequently and recently changed code to build an expertise model of developers. By using the vocabulary of software changes, lexical information about the role of elements and the kind of tasks are included in our expertise model.

Siy et al. [39] present a way to summarize developer work history in terms of the files they have modified over time by segmenting the CVS change data of individual Eclipse developers. They show that the files modified by developers tend to change significantly over time, though most of the developers tend to work within the same directories. This agrees with our observation regarding the decay of vocabulary.

Gousios et al. [17] present an approach for evaluating developer contributions to the software development process based on data acquired from software repositories and collaboration infrastructures. However, their expertise model does not include the vocabulary of software changes and is thus not queryable using the content of bug reports.

Alonso et al. [1] describe an approach using classification of the file paths of contributed source code files to derive the expertise of developers. It remains open if file paths would be sufficient to classify bug reports as an application of the model is not provided.

Schuler and Zimmerman [38] introduce the concept of usage expertise, which manifests itself whenever developers are using functionality, *e.g.* by calling API methods. They present preliminary results for the Eclipse project indicating that usage expertise is a promising complement to implementation expertise (such as our expertise model). Given these results, we consider as future work to extend our expertise model with usage expertise as well.

Mailing lists are a valuable source of developer expertise. It has been shown that the frequency with which software entities (functions, methods, classes, *etc.*) are mentioned in mail correlates with the number of times these entities are included in changes to the software [33]. It has further been shown that approx. 70% of the vocabulary used in source code changes is found in mailing lists as well [9]; we can thus estimate the impact of including mailinglists on our results. Given these results, we consider as future work to including vocabulary from mailing lists in our expertise model. We expect that this will considerably improve our results, since Information Retrieval approaches are known to perform the better the more data is available [6].

Hindle et al. [19] perform a case study that includes the manual classification of large commits. They show that large commits tend to be perfective while small commits are more likely to be corrective. Commits are not normalized in our expertise model, thus the size of a commit may affect our model. However, since large commits are rare and small commits are common, we can expect our expertise model to equally take perfective and corrective contributions into account. However, it would be interesting for future work to introduce weightings for commit sizes to our model.

Hill et al. [18] present an automatic mining technique to expand abbreviations in source code. Automatically generated abbreviation expansions can be used to enhance software maintenance tools that utilize natural language information, such as our approach. If the same abbreviations are used in both souce code and bug reports then our approach is not affected by this issue, nevertheless it would be interesting to include automatic abbreviation expansion as future work.

Currently our expertise model is limited to the granularity of committed software changes. A more fine grained acquisition of the model could be achieved by using a change-aware development environment that records developer vocabulary as the software is written, as suggested by Omori and Maruyam [32] and Robbes and Lanza [37].

Minto and Murphy's Emergent Expertise Locator (EEL) [29] recommends a ranked list of experts for a set of files of interest. The expertise is calculated based on how many times which files have been changed together and how many times which author has changed what file. They validate their approach by comparing recommended experts for files changed during a bug fix with the developers commenting on the corresponding bug report, stating that they "either have expertise in this area or gain expertise through the discussion". In our evaluation, we also consider the people commenting on a bug report as possible experts.

Anvik and Murphy evaluate approaches that mine implementation expertise from a software repository or from a bug tracking system [5]. One approach of each type is used to recommend experts for a bug report. For the approach that gathers information from the software repository, a linkage between bug reports and source files is required.

Developers who ever have contributed to the files that were touched during the resolution of bug reports similar to the one to fix are taken as possible experts. For the approach that mines the reports itself, amongst others, the commenters of the reports (if they are developers) are estimated as possible experts. Both approaches disregard inactive developers. Both recommendation sets are then compared to human generated expert sets for the bug report. The authors conclude that depending on the requirements, either approach was best.

Work using author-topic models includes: Linstead et al., who apply author-topic models on Eclipse [25]. They address the effectiveness of the model for mining developer contributions and similarities. They extract words from source code files to generate a word-document matrix. An additional author-document matrix is built from a bug database where information about developers and the source files they have changed is stored. Baldi et al. use topic models to mine aspects [7].

Lexical information of source code has previously been proven useful for other tasks in software engineering, such as: identifying high-level conceptual clones [26], recovering traceability links between external documentation and source code [2], automatically categorizing software projects in open-source repositories [21], visualizing conceptual correlations among software artifacts [23, 24], and defining cohesion and coupling metrics [27, 36].

2.2 Bug Triage

In this section, we present existing work related to bug triage as well as recommendation systems specific to bug triage.

Weiss et al. [41] present an approach that automatically predicts the fixing effort, *i.e.*, the person-hours spent on fixing a software issue. Given an issue and an issue tracking system, they use text similarity techniques to identify the most closely related issue reports and use their average fixing time as a prediction.

Bettenburg et al. [10] present an approach to split bug reports into natural text parts and structured parts, *i.e.* source code fragments or stack traces. Our approach treats both the same, since counting word frequencies is applicable for natural-language text as well as source code in the same way.

Cubranic and Murphy [40] propose to use machine learning techniques to assist in bug triage. Their prototype uses supervised Bayesian learning to train a classifier with the textual content of resolved bug reports. This is then used to classify newly incoming bug reports. They can correctly predict 30% of the report assignments, considering Eclipse as a case study.

Anvik et al. [4] build developers' expertise from previous bug reports and try to assign current reports based on this expertise. Previous reports involving developers with a too low bug fixing frequency or involving developers not working on the project anymore are filtered. They assign bug reports from a period of less than half a year. To find possible experts for their precision and recall calculation, they collect the developers who worked on the modules (*i.e.* (sub-)components) that contained the files that were touched during the bug fix in the source code (by looking for the corresponding bug ID in the change comments). They reach precision levels of 57% and 64% on the Eclipse and Firefox development projects respectively while calibrating their approach. However, they only achieve around 6% precision on the Gnu C Compiler project which is used to validate their approach. The highest recall they achieve is on average 10% (Eclipse), 3% (Firefox) and 8% (gcc). Please note that their results are not directly comparable to ours, as they use a different setting of possible experts for their precision and recall calculation.

Canfora and Cerulo [11] propose an Information Retrieval technique to assign developers to bug reports and to predict the files impacted by the bug's fix. They use the lexical content of bug reports to index source files as well as developers. They do not use vocabulary found in source files, rather they assign to source files the vocabulary of related bug reports. The same is done for developers. For the assignments of developers, they achieve 30%–50% top-1 recall for KDE and 10% to 20% top-1 recall for the Mozilla case study.

Similar to our work, Di Lucca et al. use Information Retrieval approaches to classify maintenance requests [14]. They train a classifier on previously assigned bug reports, which is then used to classify incoming bug reports. They evaluate different classifiers, one of them being a term-documents matrix using cosine similarity. However, this matrix is used to model the vocabulary of previous bug reports and not the vocabulary of developers.

Podgurski et al. classify failure reports [34]. They cluster profiles of failed executions in order to classify new failures and to diagnose common defect sources.

Baysal et al. present a framework for automated assignment of bug fixing tasks [8]. The system automatically assigns a new arriving bug report to the appropriate developer considering his or her expertise, current workload, and preferences. They infer the developer's expertise by tracking the history of the bugs previously resolved by this developer. The developer's preferences for fixing certain types of bugs are determined by preference elicitation methods, *i.e.* developers are expected to provide feedback for every bug they fix. We consider taking the workload and the preferences of developers into account as well in future work.

Concurrently to our work, Kagdi and Poshyvanyk present an approach to recommend a list of developers to assist in performing software changes given a textual change request [20]. Similar to us, they do not need previous bug reports or an explicit linkage of bug reports and source code files to train their system. First, a technique based on Information Retrieval (more precisely, LSI) is used to locate the relevant units of source code, e.g., files, classes, and methods, to a given change request. Then these units are fed to a technique that recommends developers based on the frequency of their contributions to the units of source code. The authors provide no explicit results as they have not validated their approach with systematic studies yet.

Chapter 3

The Develect Expertise Model

In this chapter, we first describe our expertise model in general and then present its construction and application in detail.

Given the natural language description of a bug report, we aim to find the developer with the best expertise regarding the content of the bug report. For this purpose, we model developer expertise using their source code contributions.

Developers gain expertise by either writing new source code or working on existing source code. Therefore, we use the vocabulary of source code contributions to quantify the expertise of developers. Whenever a developer writes new source code or works on existing source code, the vocabulary of mutated lines, surrounding lines and of the developer's comment message describing the mutation is added to the expertise model. These lines are extracted from the version control system using the `diff` command[1] and the system's `log` (for the comment message).

Natural language documents differ from source code in their structure and grammar, thus we treat both kinds of documents as unstructured bags of words. We use Information Retrieval techniques to match the word frequencies in bug reports to the word frequencies in source code. This requires that developers use meaningful names *e.g.* for variables and methods (which is the case given modern naming conventions [23]) and that the same words are used in bug reports (see also chapter 7).

Given a bug report, we rank the developers by their expertise regarding this bug report, which is given by the lexical similarity of their vocabulary to the content of the bug report.

[1]Commonly version control systems provide a `diff` command which shows changes between files in the *unified diff* format

3.1 Extracting the Vocabulary of Developers from Version Control

To extract the vocabulary of a specific developer we must know which parts of the source code have been authored by which developer. Therefore, we extract the developers' vocabulary using the version control system of the software repository. In this section, we first briefly describe how we handle different notions of a revision in different version control systems. Then we present the `diff` and the `log` of a revision and describe how we use them to collect the developers' vocabulary.

We extract the vocabulary in two steps, first building a CHRONIA model of the entire repository [16] and then collecting word frequencies from the `diff` and `log` of each revision. Depending on the particular versioning system, it can require an additional effort to identify revisions. For example, in CVS version numbers are associated with single files instead of the entire repository. Therefore, we build a Chronia model of the repository which allows us to access file revisions made within a specific time frame (more precisely, using a sliding time-window of 2 minutes [42]) as one large, *combined* revision.

3.1.1 The Diff

The `diff` command provides a line-by-line summary of the changes between two versions of the same file. The diff of a revision summarizes the changes made to all files that changed in that revision—or in that combined revision, for version control systems like CVS—of the repository. In detail, a `diff` consists of a head and a body. The head contains information about the two versions being compared, such as the date and time they were committed, their revision numbers and their file paths. The body consists of one or multiple change hunks ("chunks"), each showing the changes within particular parts of the files. A chunk begins with a line describing its position within the two files. The subsequent lines are each of one of these three types:

- Added line: A line that was newly added to the file; marked with a '+' at its beginning.
- Removed line: A line that was removed from the file; marked with a '-' at its beginning.
- Context line: A line that shows the context of the changed lines. Such a line was not changed at all and begins with a space character.

There is no special type to represent a changed line. A change in a line results in the deletion of the line before the change and the addition of the line after the change. Listing 3.1 shows a diff from the Eclipse source code repository.

```
Index: org.eclipse.debug.core/core/org/eclipse/debug/core/DebugPlugin.java
diff -u org.eclipse.debug.core/core/org/eclipse/debug/core/DebugPlugin.java:1.63 org.
    eclipse.debug.core/core/org/eclipse/debug/core/DebugPlugin.java:1.65
```

```
--- org.eclipse.debug.core/core/org/eclipse/debug/core/DebugPlugin.java:1.63    Fri May
     23 11:25:29 2003
+++ org.eclipse.debug.core/core/org/eclipse/debug/core/DebugPlugin.java Wed May 28
     13:38:29 2003
@@ -32,7 +32,7 @@
 import org.eclipse.debug.core.model.IProcess;
 import org.eclipse.debug.core.variables.ContextLaunchVariableRegistry;
 import org.eclipse.debug.core.variables.IContextLaunchVariableRegistry;
-import org.eclipse.debug.core.variables.ISimpleVariableRegistry;
+import org.eclipse.debug.core.variables.ISimpleLaunchVariableRegistry;
 import org.eclipse.debug.internal.core.BreakpointManager;
 import org.eclipse.debug.internal.core.DebugCoreMessages;
 import org.eclipse.debug.internal.core.ExpressionManager;
@@ -367,7 +367,7 @@
          *
          * @return the registry of simple launch variables
          */
-       public ISimpleVariableRegistry getSimpleVariableRegistry() {
+       public ISimpleLaunchVariableRegistry getSimpleVariableRegistry() {
                if (fSimpleVariableRegistry == null) {
                        fSimpleVariableRegistry = new SimpleVariableRegistry();
                }
@@ -473,9 +473,6 @@
                if (fEventListeners != null) {
                        fEventListeners.removeAll();
                }
-               if (fSimpleVariableRegistry != null) {
-                       fSimpleVariableRegistry.storeVariables();
-               }
                setDefault(null);
                ResourcesPlugin.getWorkspace().removeSaveParticipant(this);
        }
Index: org.eclipse.debug.core/core/org/eclipse/debug/core/ILaunchManager.java
```

Listing 3.1: An excerpt from a diff of a revision of the Eclipse repository.

The identifier names (*i.e.* names for classes, methods, attributes, variables, *etc.*) and comments that appear on these lines give us evidence of the contributor's expertise in the system. Thus, we add the word frequencies in a revision's `diff` to the expertise model of the contributing developer. This includes all types of lines, assuming that any kind of change (even deletions) requires developer knowledge and thus familiarity with the vocabulary. Note that the `diff` itself provides no information identifying the developer performing the change; this information has to be retrieved from the corresponding `log`.

The word frequencies are extracted as follows: the lines are split into sequences of words, which are further split at adjacent lower- and uppercase letters to accommodate the common *camel case* naming convention. (For example, the word "camelCase" would be split into the words "camel" and "case".) Next, stopwords are removed (*i.e.* common English words such as *the*, *and*, etc.). Eventually stemming [35] is applied to remove the grammatical suffix of words.

3.1.2 The Log

The `log` command provides information for a set of files or revisions. The `log` of a file lists the file's version history, the `log` of a revision lists information for that particular revision (or all revisions contained in that combined revision, for version control systems like CVS). In detail, a log contains information about a (file-)version such as its version number, the time it was committed, the (login-)name of the contributing developer and his/her comment message regarding that version. Listing 3.2 shows a log from the Eclipse source code repository.

The comment message usually is in prose, whereas the lines from the diff mostly contain source code. As the purpose of the message is to comment the contributions made, it most likely requires the same knowledge and thus a similar vocabulary. Therefore, the comment message associated with a revision is processed in the same way as the lines of the `diff`, and added to the expertise model of the contributing developer as well.

```
=============================================================================

RCS file: /cvsroot/eclipse/org.eclipse.swt/Eclipse SWT/win32/org/eclipse/swt/graphics/
     Path.java,v
head: 1.39
branch:
locks: strict
access list:
symbolic names:
        v3450: 1.36
        v35178: 1.39
        .
        .
        .
        v3123a: 1.1
        v3123: 1.1
keyword substitution: kv
total revisions: 39;    selected revisions: 39
description:
----------------------------
revision 1.39
date: 2008-10-23 22:17:38 +0200;  author: fheidric;  state: Exp;  lines: +3 -1;  commitid
     : 47a44900dbe24567;
removing bad fix
----------------------------
revision 1.38
date: 2008-10-23 21:11:37 +0200;  author: fheidric;  state: Exp;  lines: +1 -3;  commitid
     : 3f9524900cc684567;
251818 - drawString works improperly after drawing any primitives
----------------------------
revision 1.37
date: 2008-10-09 23:57:11 +0200;  author: fheidric;  state: Exp;  lines: +1 -1;  commitid
     : 1bb8748ee7e344567;
Bug 248551 - BidiLevel is improperly set to -1
----------------------------

.
.
.
----------------------------
revision 1.1
date: 2005-02-12 01:04:54 +0100;  author: silenio;  state: Exp;
advanced graphics API (initial)
=============================================================================
```

Listing 3.2: An excerpt from the log of the Eclipse repository.

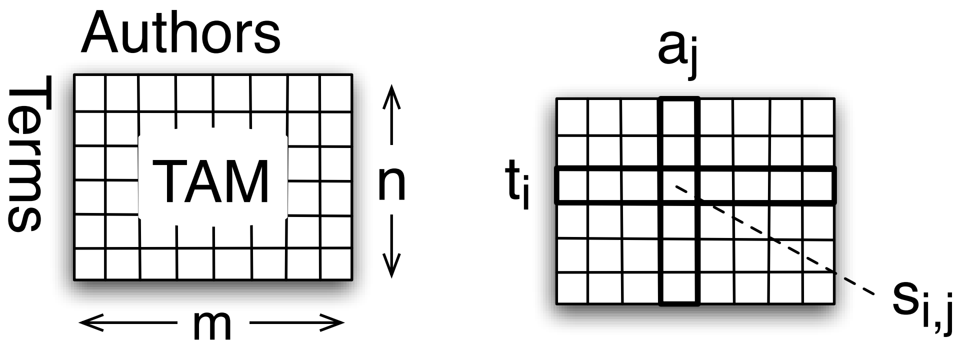

Figure 3.1: Term-Author-Matrix.

3.2 Modeling the Expertise of Developers as a Term-Author-Matrix

We store the expertise model in a matrix that correlates word frequencies with developers. We refer to this matrix as a term-author-matrix. Although, technically it is a term-document-matrix where the documents are developers. (It is common in Information Retrieval to describe documents as bags of words, thus our model is essentially the same, with developers standing in for documents.)

The term-author-matrix has dimension $n \times m$, where n is the global number of words and m the number of contributors, that is developers. Each entry $s_{i,j}$ equals the frequencies of the word t_i summed up over all source code contributions provided by the developer a_j.

We have found that results improve if the term-author-matrix is weighted as follows:

- *Decay of Vocabulary.* For each revision, the word frequencies are weighted by a decay factor that is proportional to the age of the revision. In the Eclipse case study, best results are obtained with a weighting of 3% per week (which accumulates to 50% per half year and 80% per annum). Please refer to chapter 5 for a detailed discussion.

3.3 Assign Developers to Bug Reports Regarding Their Expertise

To assign developers to bug reports, we use the bug report's textual content as a search query to the term-author-matrix. Our current tool supports the extraction and the mining of bug reports from the bug tracking system Bugzilla[2]. However, our approach is able to handle any textual representation of bug reports, even raw text.

In the following, we first present a Bugzilla bug report and describe the extraction of its vocabulary. Then we cover the calculation of the lexical similarity between that vocabulary and the vocabulary of a developer and how this similarity is used for the assignment of developers to bug reports.

3.3.1 The Bugzilla Bug Report

Given a Bugzilla bug report, we count the word frequencies in its textual content. A Bugzilla bug report consists of multiple fields describing the occurred bug, such as (amongst others) computer platform, operating system, specific product and version that was used, as well as a short and a long textual description of the bug. Comments, which are often added as part of a discussion, appear as further long descriptions. Listing A.1 shows a Bugzilla bug report in the XML format. Since Information Retrieval approaches are known to perform better the more data is available, we extract the vocabulary of practically all of these fields. This extraction is done in the same way as previously described for the `diff` and the `log` (removing stopwords, stemming, *etc.*). In particular we process both short and all long descriptions. (For threats to the validity of our approach, see chapter 5.) The only fields we disregard are attachments (which are Base-64 encoded) such as attached images, as well as fields that refer to persons (reporter "reporter", assignee "assigned_to", commenters "who").

3.3.2 The Assignment

From the bug report's extracted word frequencies, we create a *term vector* that uses the same word indices as the term-author-matrix. That is, we create a vector v of length n where v_i equals the frequency of the word t_i in the bug report.

We compute the lexical similarity between two term vectors by taking the cosine of the angle between them. The similarity values range from 1.0 ($\cos 0°$) for identical vectors to 0.0 ($\cos 90°$) for vectors without shared terms. (Negative similarity values are not possible, since word-frequencies cannot be negative.) This value is easily computed by first scaling each of the two vectors to the unit length (*i.e.* a length of 1) and then taking their dot product. The length of the term vectors should not matter when being tested

[2] http://www.bugzilla.org

for lexical similarity. That is, a vector built from a large document should be similar to a vector obtained from a small document of similar vocabulary. This is why we take the angle, and not for example the Euclidean distance, as the measure of lexical similarity.

We compare the term vector of the bug report with the term vectors of all developers (*i.e.* the columns of the term-author-matrix) and create a ranking of developers according to that comparison. For the assignment of bug reports to developers, a suggestion list of the top-k developers with the highest lexical similarities is then provided.

We have found that the results improve if the term-author-matrix is further weighted as follows:

- *Inactive Developer Penalty.* If a developer has been inactive for more than three months, the lexical similarity is decreased by a penalty proportional to the time since his/her latest contribution. In the Eclipse case study, best results are obtained with a penalty of 0.2 per annum. Please refer to chapter 5 for a detailed discussion.

Chapter 4

Case Study: Eclipse

In this chapter, we present the evaluation of our approach using Eclipse as a case study. In chapter 6, a project from Gnome is used as an additional case study to validate the obtained results.

Eclipse[1] is a large open source software project with numerous active developers. It has been developed over several years now. Therefore, its version repository contains a great deal of source code developed by many different authors. Eclipse uses Bugzilla as its bug tracking system, storing bug reports dating back to nearly the beginning of the project. We evaluate our recommendations by comparing the top-k developers with the persons who eventually worked on the bug report.

This case study covers the Eclipse project between April 22, 2001, and November 9, 2008. The source code contributions of Eclipse are stored in a CVS repository[2], the bug reports in a Bugzilla database[3]. This represents almost eight years of development, including 130,769 bug reports and 162,942 global revisions (initially obtained from CVS's file versions using Chronia's sliding time-window, later processed on a weekly basis, see section 5.3). During this time, 210 developers contributed to the project.

4.1 Setup of the Case Study

The setup of the Eclipse case study consists of two parts. The first part is about the change database where we use all changes before the actual bug report. The second part is about the bug database, there we make 10 partitions of which two are used in this case study. We process and evaluate both parts in weekly iterations as follows:

[1] http://www.eclipse.org/eclipse
[2] :pserver:anonymous@dev.eclipse.org/cvsroot/eclipse
[3] https://bugs.eclipse.org/bugs

- We create a DEVELECT expertise model based on contributions between April 22, 2001 and the last day of the previous week.
- We generate a suggestion list of the top-10 experts for all bug reports submitted in the current week.
- We evaluate precision and recall by comparing the suggested list with the developers who, between the first day of the next week and November 9, 2008, eventually worked on the bug report.

For example, for a bug report submitted on May 21, 2005 (Sat), we would train our system with all commits between April 22, 2001 and May 15, 2005 (Sun), and then evaluate the list of suggested experts against the set of developers who, between May 23, 2005 (Mon), and November 9, 2008, eventually handled the bug report.

We use systematic sampling to create 10 partitions of 13,077 bug reports (ordered by time) that span the entire time of the project. In other words, we first sort the reports by time. Then we iterate through the list and assign every 10th report to the same group. This yields 10 groups all containing bug reports similarly distributed over the entire time of the project. One partition is used as training set for the development of our approach, and another partition is used as validation set to validate our approach. We applied the approach to the validation set only after all implementation details and all calibration parameters had been finally decided on. The other partitions remain untouched for use as validation sets for future work.

In this chapter, we report on our results obtained on the validation partition #2. In chapter 5 we report on results obtained from the training partition #1 while calibrating the approach.

4.2 Precision and Recall Configurations

We evaluate our approach by comparing the suggested list of experts with the developers who eventually worked on the bug report. We report on precision and recall for different sizes of suggested lists, between $k = 1$ and $k = 10$. Precision is the percentage of suggested developers who actually worked on the bug report. Recall is the percentage of developers who worked on the bug who were actually suggested. It is typical for Information Retrieval approaches that there is a trade-off between precision and recall.

In order to decide if a developer worked on a bug report or if a person working on a report is a developer, we need a mapping between CVS logins and people mentioned in bug reports. For a detailed discussion of how we achieve this mapping and its threats to validity please refer to section 5.4.

Comparing our results to the persons who eventually worked on the bug is not optimal. For example, the person could have been assigned to the bug report by some factor other than expertise. Obtaining a better list of experts requires manual interrogation of the development team.

Getting the list of persons who eventually worked on a bug report is tricky. The *assigned-to* field does not always denote a person who eventually solved the bug report [40, 4, 5]. For example, when a bug report is handled quickly and its status changes from *new* directly to *closed/fixed*, the report might not have been properly assigned yet. Then the developer who closed the report or provided the fix in the comments is the one who solved the bug.

In related work, there are therefore often sophisticated heuristics applied to retrieve the real report-solving person. We only use three simple configurations. On the one hand, since we use the software repository as information source, we are able to retrieve the developers who eventually fixed the bug in the source code and can compare our suggestions directly against them. On the other hand, we seek to suggest experts for the general handling of the report. Thus, we do not necessarily need to only match the eventual bug fixing person, but also the people who understand and discuss the bug. Therefore, we compare our results against three configurations (C1–C3) of bug-related persons:

1. Developers who committed an actual bug fix to the software repository. For Eclipse, this information is not stored in the Bugzilla database, therefore we must rely on information from CVS commit messages. In the validation set, this information is provided for 14.3% of the bug reports only. This configuration evaluates how well we perform in suggesting experts who provide actual bug fixes.

2. Persons given by the *assigned-to* field or a *who* field of the bug report. That is, the eventual assignee (if this is a developer) and all developers who ever discussed the bug in the comment section of the report. This configuration evaluates how well we perform in suggesting experts who are capable of understanding and discussing the bug. Note that resolving a bug is not limited to providing code fixes; often the discussion is just as important to the resolution of the bug.

3. As in configuration #2, but additionally including the person identified by the *reporter* field, if the reporter is a developer, *i.e.* has a CVS login. This reflects the fact that bugs are sometimes resolved by the same people who find and track them.

Please refer to chapter 5 for a further discussion of the configurations listed above and their threats to validity.

4.3 Results

Figure 4.1 illustrates the results of the Eclipse case study. We compare lists of suggested persons of list size 1 to 10 with sets of bug-related persons as given by the three

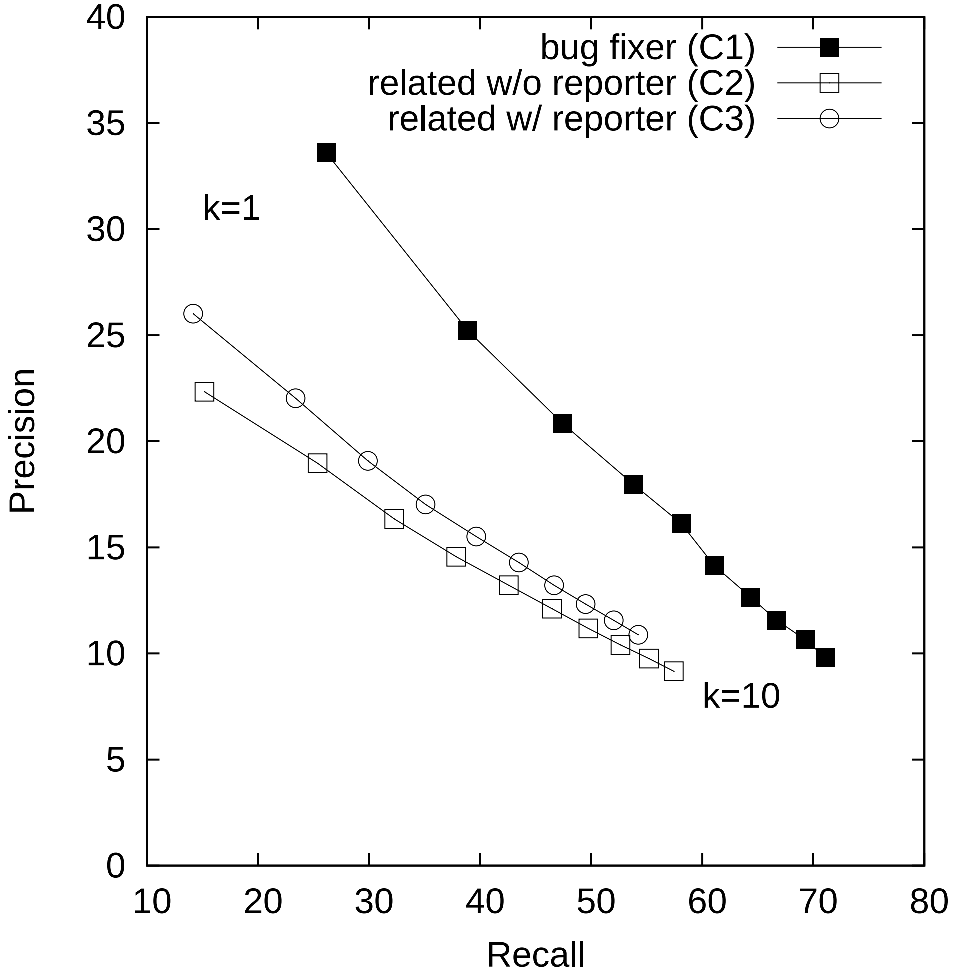

Figure 4.1: Recall and precision of the Eclipse case study: configuration C1 scores best with 33.6% top-1 precision and 71.0% top-10 recall.

configurations (C1-3) above.

The figure illustrates that recall and precision of configuration C1 are better than C2 and C3. Comparing the suggested list to the bug fixing persons (C1), we achieved the best score with 33.6% top-1 precision and 71.0% top-10 recall. This means that on average, 33.6% of the top-1 suggested experts match the actual bug fixing persons, and on average, 71.0% of the actual bug fixing persons appear amongst the top-10 suggested experts. Comparing the suggested list to all related persons (C3), we score 26.0% top-1 precision and 54.2% top-10 recall. When excluding the reporter of the bug report (C2) from the suggested list the scores are at 22.3% top-1 precision and 57.4% top-10 recall.

The fact that configuration C3 scores slightly better than C2 indicates that bug reporters are sometimes indeed experts regarding the reported bug and thus should be considered when triaging bug reports. We can thus conclude that an automatic assignment system should provide to the triaging person a suggested list of people that may include the reporter.

Chapter 5

Discussion

In this chapter, we first discuss general important threats to validity, then report on the calibration of our approach and eventually cover other dangers to its soundness.

Compared to related work, an advantage of our approach is that we do not require a record of previous bug reports to suggest experts. We only need previous reports and their related people for the evaluation of our approach. Thus, we further do not have to rely on heuristics in order to train our recommendation system with the assignees of the previous reports or the people who eventually resolved them. In addition, we are able to recommend developers who did not work on bugs previously. For example, we do not require that developers have worked on at least a certain number of resolved bug reports to be assignable. On the other hand, our approach requires at least a half to one year of versioning history in order to suggest developers.

One obvious threat to validity is the quality of our evaluation benchmark. We compare our suggested list against the developers who eventually worked on the bug report and assume that these are the top experts. But, for example, the bug report could have been assigned to a developer by some factor other than expertise. This threat is hard to counter. A better list of experts can be obtained by manual interrogation of the development team, but even this is not a golden oracle.

It is difficult to precisely and reliably evaluate the performance of our approach. This is not only caused by the suboptimal evaluation benchmark, but also influenced by additional problems that arise in the specific case studies, *e.g.* an ambiguous or incomplete mapping of login names to bug report-related people. These issues mainly affect the evaluation but only to a small degree the approach itself. Therefore, the performance of our approach is probably better than indicated by the evaluation of the case studies.

Another threat to validity is that we use all long descriptions, including comments, of a bug report as information source. This may include discussions that happened after the bug has been eventually assigned or fixed, information which is not actually

Settings	Precision	Recall
reference	19.7	42.6
added lines only	18.5	41.1
desc. fields only	16.7	37.7
with LSI	16.5	35.1
decay 0.03	20.5	43.2
decay 0.05	20.4	42.4
decay 0.10	20.5	41.5
decay 0.20	18.0	38.0
penalty 0.1	24.5	50.9
penalty 0.2	24.8	51.6
penalty 0.3	24.8	51.9
final calibration	26.2	54.6

Table 5.1: Summary of calibration of training set, for each settings top-1 precision and top-10 recall are given.

available when doing initial bug triage. This might impact the performance of our approach.

5.1 On the Calibration of DEVELECT

We used 1/10th of the Eclipse case study as a training set to calibrate our approach. The calibration results are summarized in Table 5.1.

The table lists top-1 precision and top-10 recall with comparison of the suggested list to all bug report-related persons (C3). On the first row, we list the results before calibration $(p = 19.7\%, r = 42.6\%)$, and on the last row the results of the final calibration. Please note that the final results on the training set are slightly better than the results reported in section 4.3 for the validation set. This might indicate a minor overfitting of our model to the data of the training set.

The output of the `diff` command consists of added, removed and context lines. The `log` command provides the lines of the corresponding revision's comment message. (See also section 3.1.) The reference results in Table 5.1 were achieved by weighting all lines the same, which does not have to be the optimal configuration. According to our assumption, any kind of change in the source code (even deletions) requires developer knowledge and thus familiarity with the vocabulary. However, it is not evident that a developer has *equal* knowledge about source code he/she adds, deletes, the surrounding lines of the change and the corresponding comment message. This is why we tested some other weighting configurations. One consisted of weighting the removed and the context lines less than the added lines and the comment message, which yielded values that are slightly lower than the ones of the reference result. The result shown as "added lines only" was obtained by only considering the added lines and has even lower values. Therefore we decided to not weight the lines differently.

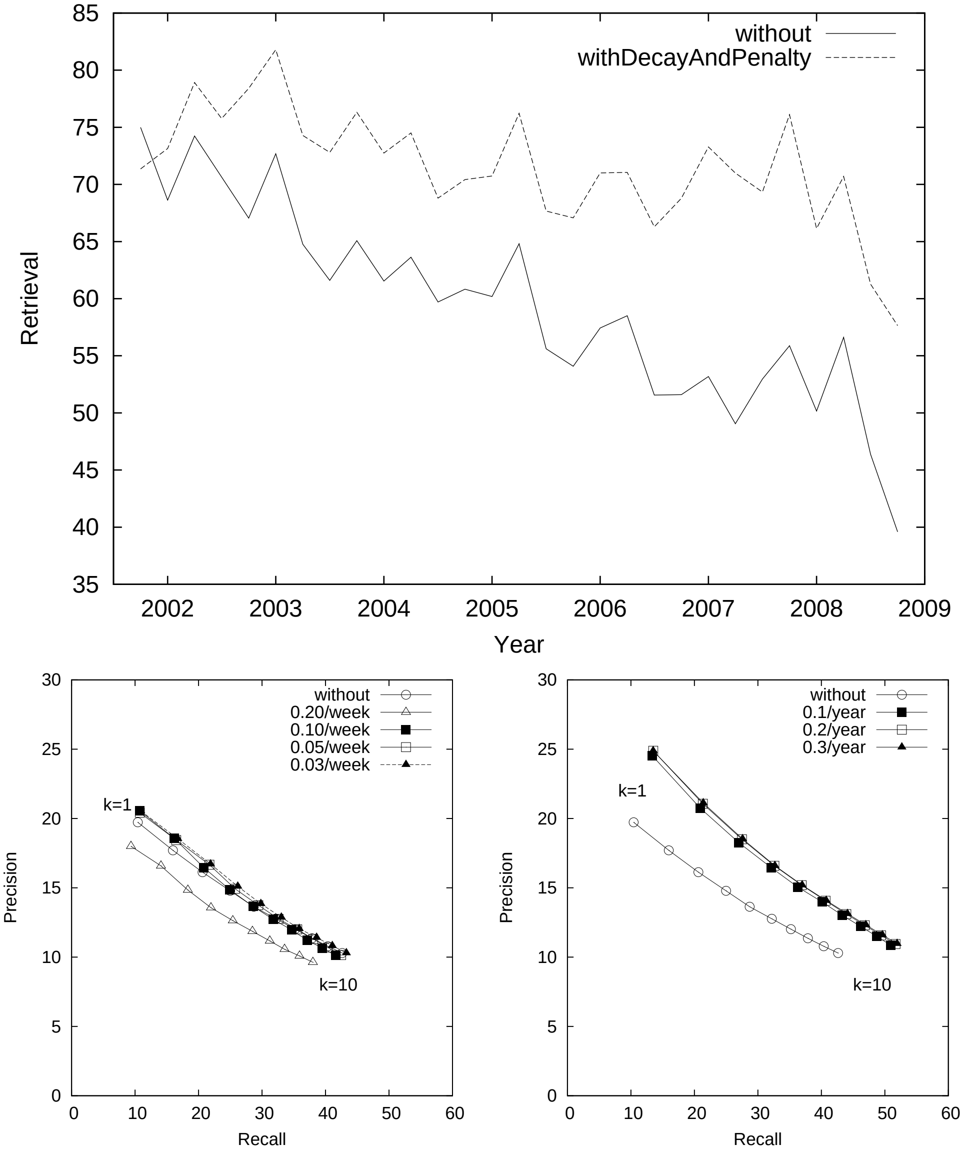

Figure 5.1: Decay of developer expertise: (top) decreasing quality of unweighted results, compared to results with *decay of vocabulary* and *inactive developer penalty* settings; (left) precision and recall for different decay of vocabulary settings; (right) precision and recall for different inactive developer penalty settings.

As Bugzilla bug reports consist of many fields, we experimented with different selections of fields. We found that taking only short and long descriptions ("desc. fields only" in Table 5.1) yields worse results than selecting all fields except those that refer to persons, or Base64 encoded attachments. This matches the fact that usually Information Retrieval techniques work better when more data is available.

We also experimented with Latent Semantic Indexing (LSI), an Information Retrieval technique typically used in search engines that detects polysemy and synonymy by statistical means [13]. However, we found that LSI yields poor results ("with LSI" in Table 5.1).

5.2 On the Decay of Vocabulary

In our experiments, we found that developer expertise decays over time. In our approach we introduced two weighting schemes to counter this effect:

- *Decay of Vocabulary.* For each revision, the word frequencies are weighted by a decay factor that is proportional to the age of the revision. Developers change their interests and by doing so change their expertise and vocabulary. To take such a shift into account, we fade the old vocabulary out bit by bit every week, so that the newest words are weighted slightly more than older ones. With time, the old words eventually fade out completely.
- *Inactive Developer Penalty.* If a developer has been inactive for more than three months, the lexical similarity is decreased by a penalty proportional to the time since his/her latest contribution to the software system. Inactive developers will most likely not resolve bugs anymore. They might still be experts for a certain part of the software system (depending on how long they have not been working on the system), but they will most probably not be involved in bug report handling anymore. In order to only recommend currently active developers (we assign bug reports during a period of eight years), developers who did not recently make a change to the software system receive a penalty. The penalty begins to apply starting from three months of inactivity because we think that all developers committing more frequently than every three months should not be penalized at all. In addition, a developer who committed four days ago would otherwise receive more penalty than one who contributed one day ago which seems not reasonable.

Figure 5.1 illustrates the effect of these settings. On the left, the unbroken curve illustrates the decreasing quality of unweighted results, whereas the dotted curve shows the results obtained with weighting. Even though results improved significantly, the quality of the weighted results still slightly decreases over time. We cannot fully explain this effect, though we suspect it could be due to the increasing complexity of Eclipse as a project. Another cause for the trend in the most recent year, *i.e.* 2008, might be that the list of persons that worked on a bug is not yet completely known to us, which may impact the evaluation. A further reason might be simply that with time, an increasing number of developers work on the project. Although the project itself gets larger as well, there might be parts with a higher density of developers who work on it. So there is an increasing number of people (eg. with shared expertise, that is similar expertise for a specific part of the system) who could be assignees, reporters or commenters of a report or who could fix a bug. This might impact the performance of suggesting the right set of experts.

In the middle of Figure 5.1, precision and recall for different *decay of vocabulary* settings are given. On the right, precision and recall for different *inactive developer penalty* settings are given.

Decay of vocabulary scores best results with a weighting of 3% per week (which accumulates to 50% per half year and 80% per annum). This shows that implementation expertise acquired one year ago or earlier does not help in assigning developers to bug reports.

The inactive developer setting scores best results with a penalty of 0.2 per annum. As a result of the penalty, the matching score of a developer who has been inactive for a year is decreased. The matching scores are the lexical similarity values (between 1.0 and 0.0). Decreasing this value by 0.1 or more is typically enough to exclude a result from the top-10 list. Interestingly, any penalty above 0.1 is better than none. The results obtained with different penalty values are almost the same.

Please note, that even though the penalty removes inactive developers from the top of the suggested list, their vocabulary is not lost. The results reported for the calibration of the penalty do not make use of vocabulary decay. If a developer becomes active again, all his/her past expertise is reactivated as well. Thus, we use a moderate penalty of 0.2 in combination with a decay of 3% as the final calibration settings.

5.3 On Grouping Diffs by Week

To cope with the size of our case study, we decided to run weekly iterations rather than fine-grained iterations per bug report and file revision. This reduced the time expenditure from over 160,000 iterations down to 394 weekly iterations.

This grouping of diffs by both author *and* week introduces the following threats to validity:

- If vocabulary is added and removed within the same week, it does not add to the developer's expertise. In the same way, if a file is added and removed within the same week, it is not taken into account at all.
- If bug reports are submitted late in the week, we might miss developers who acquired novel expertise early in the week.

- If several authors worked on the same file during the week, we cannot tell their contributions apart. In this case, we weight the word frequencies by $\frac{1}{\sqrt{n}}$, where n is the number of co-developers, and assign the weighted frequencies to all co-developers. For the Eclipse case study, this applies to 3.6% of weekly file changes.

5.4 On Other Threats to Validity

On the mapping of logins to bug report-related people. Establishing an identity relationship between CVS logins and people mentioned in bug reports is not trivial. The developer information in the CVS log is provided as a mere login name. People mentioned in a Bugzilla bug report are listed with their email address and often additionally with their first and last name. For Eclipse, the mapping between logins and names of active developers can be found on the Eclipse website[1]. However, the list of names of the former Eclipse developers does not include their corresponding logins[2]. We were able to manually map a few logins to names of alumni developers and could eventually not find names for 17.1% of the CVS logins. We had to exclude 2.7% of the bug reports completely from our evaluation because none of their related people were developers, *i.e.* we did not have a corresponding CVS login for any of their related people. We also excluded on average 31.0% of the related people in the remaining bug reports because they seemed not to be developers. As on average 3.2 people worked on a bug report, this corresponds to approximately one out of three bug report-related people.

On names and emails. People mentioned in a Bugzilla bug report are listed with their email address and often additionally with their name. In the Eclipse case study, we preferred to match logins with the names over matching with the emails. This is mainly because there already was a linkage available between logins and names of active developers. On the contrary, in the Gnome case study is no unambiguous mapping provided and we match logins with email addresses (see chapter 6). A further advantage of names is that they are rather unique, *i.e.* one single person normally has one single name, whereas a developer could for instance use multiple email addresses in bug reports (which is rather unlikely though, but possible). To improve our results, we could collect the emails which appear together with the names to nevertheless include bug report-related people with no listed name. (For example, all people appearing in the *cc* field of the bug report—a field containing people who want to be notified of the report's changes—are listed without their name.) We could probably further improve our results by matching names in a fuzzy way to address misspelled names and encoding issues (*e.g.* with umlauts). Emails are probably listed automatically which would prevent such problems. An additional threat to validity

[1] http://www.eclipse.org/projects/lists.php
[2] http://www.eclipse.org/projects/committers-alumni.php

is that people with the same name are assumed to be the very same person and thus handled incorrectly.

On retrieving the real bug fixing person. In configuration C1 we compare our suggestions against the developers who committed an actual bug fix to the repository. We retrieve these developers by looking for appropriate notes in the comment messages of their commits. Often, developers state a note like "Bug 248551 - BidiLevel is improperly set to -1" (taken from Listing 3.2) as comment message of their contribution, indicating the fact that it is a bug fix and providing the ID-number of the corresponding bug report. Sometimes, the same bug is referenced in multiple commits by possibly different developers. Thus, we look for the term "bug" in the comment messages, regard subsequent numbers as ID-numbers of bugs being fixed and remember them together with the corresponding developers. We do not consider numbers without that term or before it in the messages, even though bug fixes are sometimes referenced like that as well. We prefer high precision over high recall in this case, *i.e.* we want to be sure that the retrieved references really are ones and thus accept that we might miss some other correct references. Especially, since we still got enough remaining bug reports (more precisely, 1,872) for the validation when we do not consider those for which we did not find a reference in a comment message.

On login name multiplicity. There seem to be developers who have multiple CVS login names. We assume that they do not use them concurrently (at least not within the same project), but rather some day changed their old login to a new one. This influences our expertise model, since the logins are treated independently and the developer's vocabulary is distributed amongst them. This separation should have an impact in the year in which the change took place only, since the old vocabulary (which is separated because of the change) is subject to decay and will fade out over time. However, we could possibly improve the performance of our approach by unifying the vocabulary of the developer's different logins.

On source code copying. Information about source code file copying is not available in CVS. Bulk renaming of files appears in the change history of CVS as bulk removal of files followed by bulk addition of files. Given our current implementation of DEVELECT, this may lead to an incorrect acquisition of developer knowledge, since the entire vocabulary of the moved files is assigned to the developer who moved the files. We are thus in good shape to further improve our results by using a copy pattern detection approach [12].

Chapter 6

Case Study: Gnome Evolution

In this chapter, we present the "Evolution" project from Gnome[1] as an additional case study to validate the results obtained from the study on Eclipse, chapter 4. As Eclipse, Gnome is a large open source software project and has numerous active developers.

This case study covers the "Evolution" subproject of Gnome between Jan 1, 2005, and February 28, 2009. The source code contributions of Gnome are stored in a SVN repository[2], the bug reports in a Bugzilla database[3]. This represents slightly more than four years of development, including 42,891 bug reports and 9133 revisions. During this time, 217 developers (or more precisely, SVN logins) contributed to the project.

Similar to the study on Eclipse, we evaluate our recommendations in weekly iterations by comparing the top-k suggested developers with the developers who eventually worked on the bug report. For example, for a bug report submitted on May 23, 2007 (Wed), we would train our system with all commits between Jan 1, 2005 and May 20, 2007 (Sun), and then evaluate the list of suggested experts against the set of developers who, between May 28, 2007 (Mon), and February 28, 2009, eventually handled the bug report.

Our recommendation system has been developed and calibrated on a training set taken from a partition of bug reports from the Eclipse case study, and been validated on a validation set from another such partition (see chapter 4). Here we take the entire Gnome/Evolution project as an additional validation set to validate the previously obtained results. (We do so using a system that is trained on Gnome/Evolution source code contributions, of course.) To work on a similar number of bug reports, we use

[1] `http://projects.gnome.org/evolution`
[2] `http://svn.gnome.org/svn/evolution`
[3] `http://bugzilla.gnome.org`

systematic sampling with which we select 13,402 bug reports out of all available ones.

6.1 Establishing Links between SVN Logins and Persons in Bug Reports

Establishing reliable links between SVN logins and persons in bug reports proved to be the most challenging part of this case study. Contrary to the study on Eclipse, we did not find an unambiguous mapping from SVN logins to *e.g.* a first and a last name of the person behind the login. (In the Eclipse case study, for most of the logins unambiguous links to names were provided by the project and thus did not have to be established by ourselves.) We therefore had to find such mappings to developers first before we then could try to compare them to the people mentioned in the bug reports.

Contributions in Gnome are accompanied with a comment message that often contains at its beginning a name and an email address, supposably denoting the person who provided the source code. However, for the same SVN login, names and email addresses in the comment message vary. That is, contributions by different persons are committed using the same login and thus the retrieved links between SVN logins and persons are ambiguous.

Concerning the bug reports, we collect for each its commenters, reporter and possible assignee; in this case study, we further refer to them as the bug report-related people. These people are referenced in the bug report with an email address. It is not clear whether some people use different email addresses for handling bug reports. In order to work on a bug report, one has to register beforehand to the bug tracking system by providing an email address. We assume that not many people have registered with multiple email addresses and thus assume that different email addresses in a bug report denote different persons.

To address the ambiguity of the links, we use three different link resolution configurations; the first two based on different assumptions, the third being a combination of both.

Configuration A) We treat each SVN login as a unique person, thus ignoring that the different email addresses found in SVN commit messages might belong to different persons. It is possible that the different email addresses are due to external patches that are committed by one of the developers. We assume that this is only done if the developer him/herself is knowledgeable in the patched code, and thus adding this contribution to his/her knowledge is reasonable.

For every commit, we assign the words to the login and collect the full name and email (if existing). Logins which never had a name or email in the comment message of one of its commits are discarded. For a given bug report, we find the SVN logins with the k

highest lexical similarities. We try to match one of their email addresses with the bug related people.

For example, given we retrieve login `joe` who used in his commit messages the emails E_1 and E_2 and given a bug report with reporter email E_3 and assignee email E_1, the precision is 1, since one of the emails of `joe` matches one of the related people. Recall is 0.5, since only one of the related persons is matched by the login.

Configuration B) We assign the commits to the names in their comment messages. If there is no name in the comment of the commit, we cannot tell with certainty who actually authored this contribution and discard it. For all names, the emails appearing with them are collected. For a given bug report, we then find the names with the k highest lexical similarities and try to match one of their email addresses with the bug related people.

Configuration C) We assign the commits to the names in their comment messages or the logins. If there is a name in the comment of the commit, the words are assigned to that name. Otherwise, the words are assigned to the login. For all names and logins, the emails appearing with them are collected. We discard logins for which no email addresses have been found. For a given bug report, we then find the names *and* the logins with the k highest lexical similarities and try to match one of their email addresses with the bug related people.

In all configurations, bug related people are only considered if they are developers, *i.e.* if their email address appeared in at least one commit comment message.

6.2 Results

We report on precision and recall for every configuration for sizes of suggested lists between $k = 1$ and $k = 10$. The calculation of precision and recall is done similarly to the Eclipse case study (see section 4.2).

Figure 6.1 illustrates the results of this case study. For each configuration, we provide two data series: once comparing our suggestions to the bug fixing persons (who are retrieved by looking for corresponding comment messages), and once comparing our suggestions to all bug report related persons (*i.e.* assignee, commenters and reporter). Recall and precision of configurations A and C are considerably better than those of configuration B. For configurations A and C, results are better when compared against all bug report related people rather than when compared against bug fixers. Only in configuration B comparing against bug fixers scores better with small suggestion list sizes.

Given the assumptions of configuration A, *i.e.* that any knowledge committed by the same login belongs to one person, we score best with 24.5% top-3 precision and 82.4%

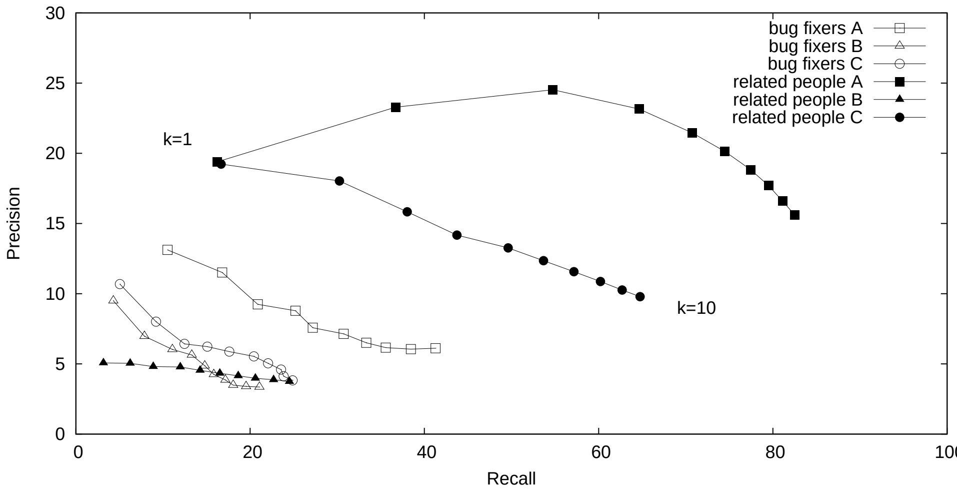

Figure 6.1: Recall and precision of the Gnome/Evolution case study: configuration B with 19.2% top-1 precision and 64.7% top-10 recall are the most reliable results.

top-10 recall. When establishing links using names only (configuration B) we achieve 9.5% top-1 precision and 21.0% top-10 recall. The combined approach, *i.e.* establishing links using names and falling back to logins if no name available (configuration C), scores with 19.2% top-1 precision and 64.7% top-10 recall.

6.3 Discussion

It might be that Gnome has a strict commit policy in a way that only very few developers are allowed to file in changes to the source code. Everyone else then provides these core-developers the source code for *e.g.* bug fixes or other contributions. If they also accept source code outside their expertise, then configuration A gives biased results. However, some login names are quite reliably assignable to their actual owners (*e.g.* some logins correspond directly to some email aliases). Even people with such an own SVN login appear with their name and email address in the comment message of another clearly assignable login, which means that even the core-developers provide each other source code to check in. This indicates that a strict commit policy is not the only reason for that behavior. One might also think of pair-programming, but then the comment message wouldn't necessarily need the details of the non-owner; those of the owner of the login would be sufficient, since he/she co-authored the contribution.

Configuration B has poor results. This is most likely due to the loss of important contributions. For this configuration, we only collect a commit if it is accompanied with a name and an email address (most likely of the actual author). But there are a lot of commits without this information, for which reason we lose potentially important vocabulary which might negatively impact the results.

The results of configurations A and B provide an upper and a lower bound for the hypothetical results achieved with perfectly established links between SVN logins and persons. Given the results of configuration C, which uses the approach of configuration B with fallback to configuration A if no name given, we can presume that the actual precision and recall of our approach are more towards the upper bound.

When we assume that configuration C has the most reliable results, we achieve for this case study 19.2% top-1 precision and 64.7% top-10 recall. This is not as good as the results from the Eclipse case study, where we obtained 33.6% top-1 precision and 71.0% top-10 recall. However, considering the difficulties and uncertainties with the mapping between the source code repository and the bug tracking system, the newly obtained results are decent and confirm the validity of our approach.

Chapter 7

Vocabulary of Source Code and Bug Reports

In this chapter, we will briefly investigate the differences and commonalities of the vocabulary used in bug reports and the vocabulary used in source code to support our assumption of both vocabularies sharing domain-specific words. For this purpose, we created two text corpora. One representing the vocabulary of bug reports by collecting the total vocabulary of the 13,077 bug reports from the Eclipse validation partition #2. The other corpus representing source code vocabulary by collecting the total vocabulary of all developers from the Eclipse expertise model. (This model was built from the Eclipse contributions of eight years of development). We used log-likelihood ratio to compare the two corpora [22]. The results are illustrated on the subsequent pages. Please keep in mind that we removed stopwords and applied stemming, therefore some common English (non-discriminative) words may be missing and some words might lack their grammatical suffix.

Figure 7.1 shows the most frequent occurring terms from the vocabulary of the bug reports and the source code contributions. The terms are ordered by their log-likelihood value. This means that the top-most words have the highest probability to occur in a bug report while the bottom-most ones have the highest probability to occur in source code. Words in the middle occur with equal probability in both bug reports and source code. To make the center (*i.e.* the region with equal probability of occurrence) more visible, the saturation of red or blue color indicates the likeliness of a word to occur in bug reports or source code, respectively. The size of a word denotes the frequency of its occurrence in both bug reports and source code. Figure 7.2 and Figure 7.3 are non-colored. Instead, the darkness of a word indicates its frequency of occurrence in only bug reports or source code, respectively.

On the bottom of the list appear Java keywords and types such as “return”, “public”, “int” and “null”. But also the HTML tags and keywords “href”, “div” and “nbsp” have a high probability to occur in source code contributions. Since HTML-files were also committed to the source code repository, this seems reasonable. Especially when noted that they are considerably smaller and therefore occur less frequent; yet, when they do occur, then most probably in source code (and not in bug reports). On the top of the list appear words like “attach”(ment), “fix”, “problem”, “verifi”(/y) and “bug” which seem typical when thinking of words to be commonly used in bug reports.

The three words that appear most often throughout both bug reports and source code contributions are “eclipse”, “java” and “org”. All these words appear one or more times in practically every source code file, since most package names start with “org.eclipse.” and appear then again in the imports together with Java core classes starting with “java.”. The words also appear in bug reports as package names (“org.eclipse”) and file endings (“.java”) when referencing classes or in stack traces, for example. In addition, the word “eclipse” occurs automatically in every bug report since it is the designated value of the *classification* field for reports related to the Eclipse project. The corpus with the source code vocabulary was built using the contributions to the project, so words from lines that changed more often (*i.e.* words that were contained in more contributions) appear more frequently in that corpus. Package names and import declarations probably change less often than other parts of the source code (yet, when they do, the three words are numerously contained), therefore the three most frequent words are not so likely to appear in source code contributions but most likely to appear in bug reports.

The words with almost equal probability of occurrence in both bug reports and source code include “wizard”, “gtk” and “folder”. The word “wizard” refers in both cases most likely to a user interface element of the Eclipse IDE, as well as the word “gtk” in both cases most likely to the GUI library used in the Eclipse source code. However, it is not clear if the word “folder” denotes a folder where Eclipse source code files are stored, or if it refers to a folder managed in the Eclipse IDE. A difficulty in this data set is thus that the functional and technical domains can sometimes not be clearly separated. As another example, the word “debug” may refer to the action of tracking and fixing a bug in the Eclipse source code, or to an action or component related to the debugger included in the Eclipse IDE.

Figure 7.2 and Figure 7.3 show that many words that are most likely to appear in bug reports, *i.e.* are typical for bug reports, occur a lot in source code. On the other hand, many words that are most likely to appear in source code, *i.e.* are typical for source code contributions, do not occur a lot in bug reports. This indicates that bug reports rather use parts of the vocabulary from source code than having an own, specific vocabulary.

In summary, we can say that this brief investigation of the vocabularies used in bug reports and source code met our expectations on a reasonable distribution of the different and common (domain-specific) words between both vocabularies; though, the technical and functional domains are sometimes not easily separable. As a further application of this technique, one could for example characterize the specific expertise of each developer by comparing his/her vocabulary to the total one of all developers.

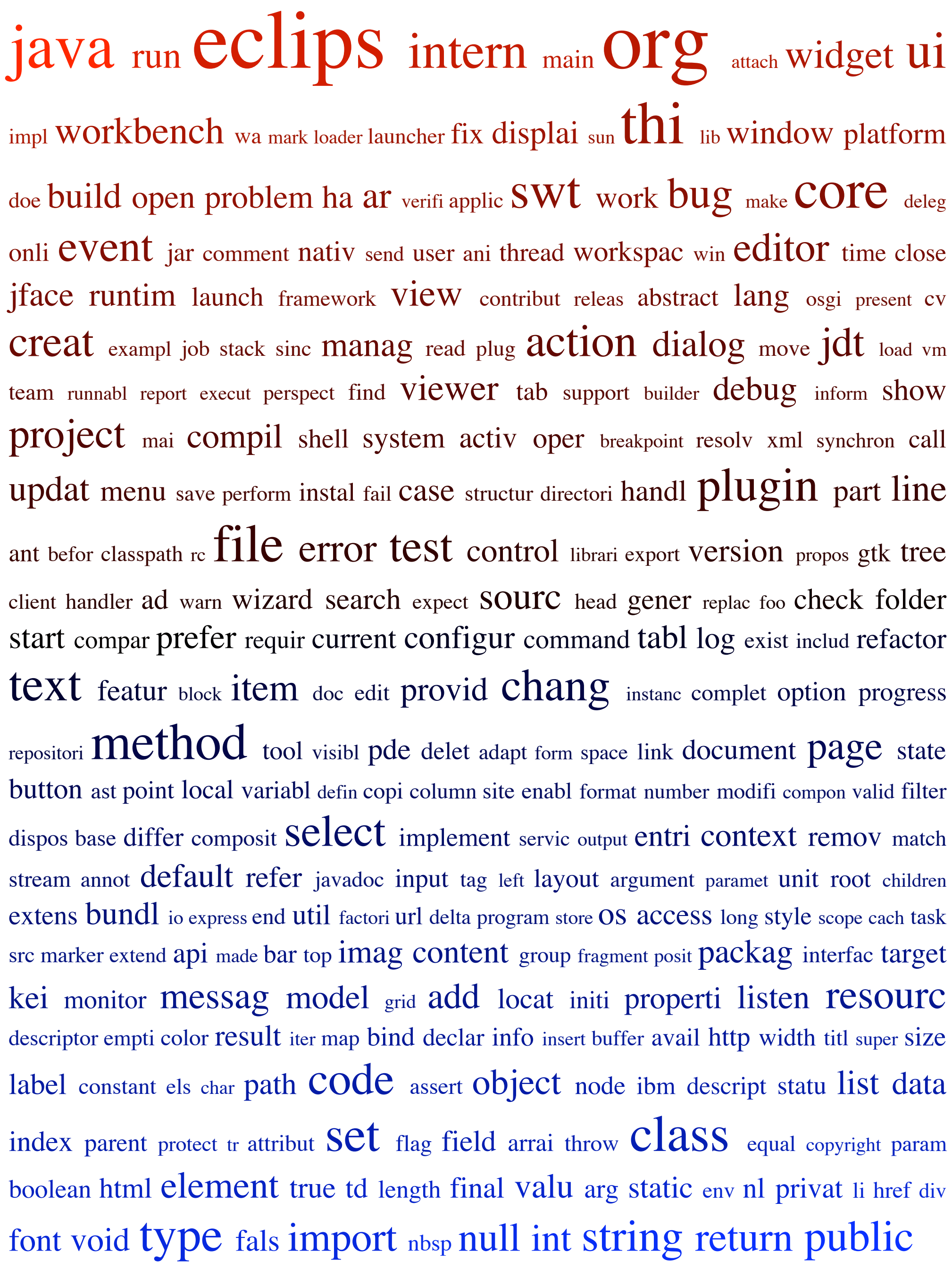

Figure 7.1: Likelihood of vocabulary in bug reports and source code.[a]

[a]This figure makes use of colors. Please make sure to obtain a color copy for a better understanding.

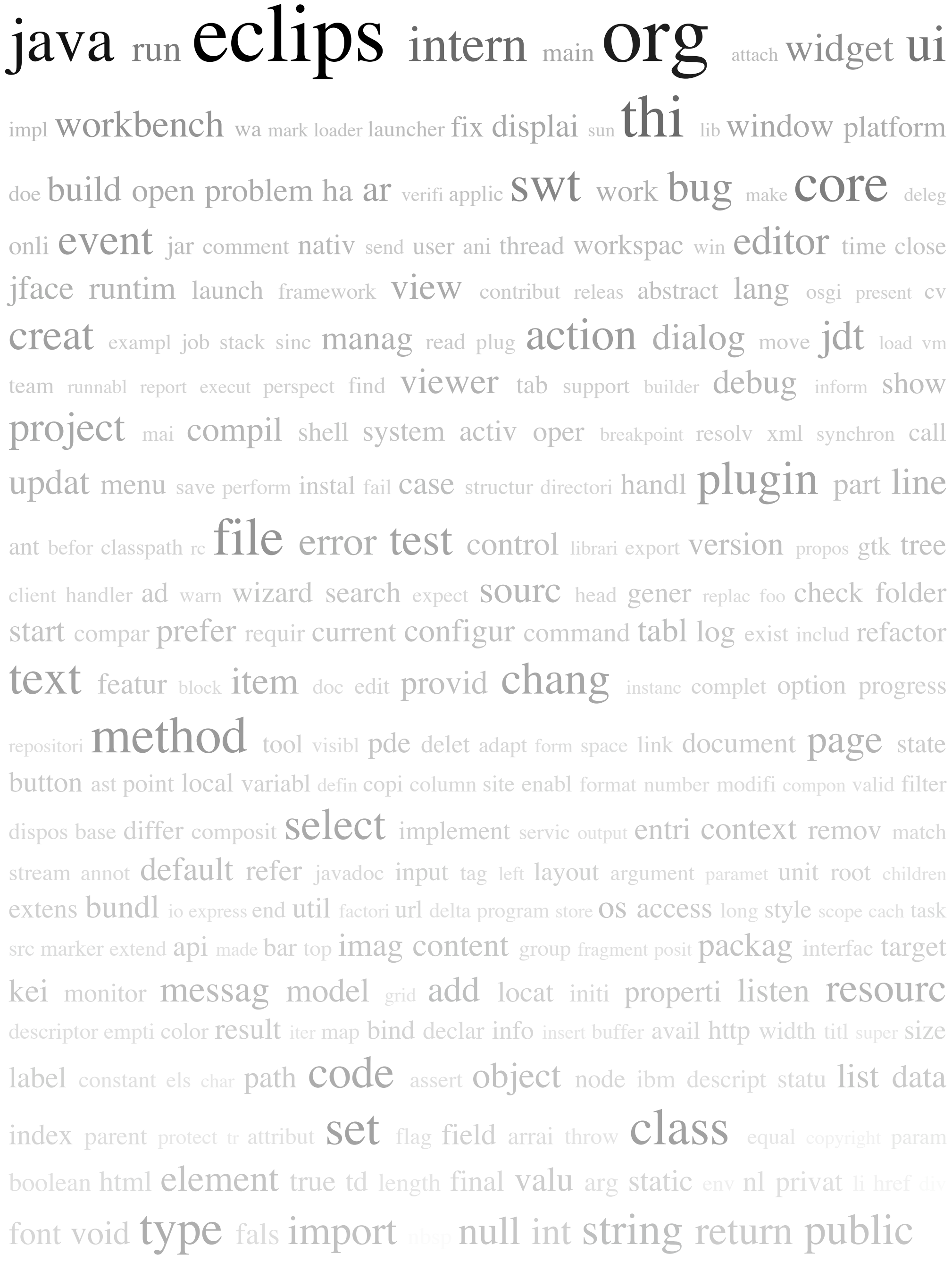

Figure 7.2: Likelihood of vocabulary in bug reports and source code with darkness indicating frequency in bug reports.

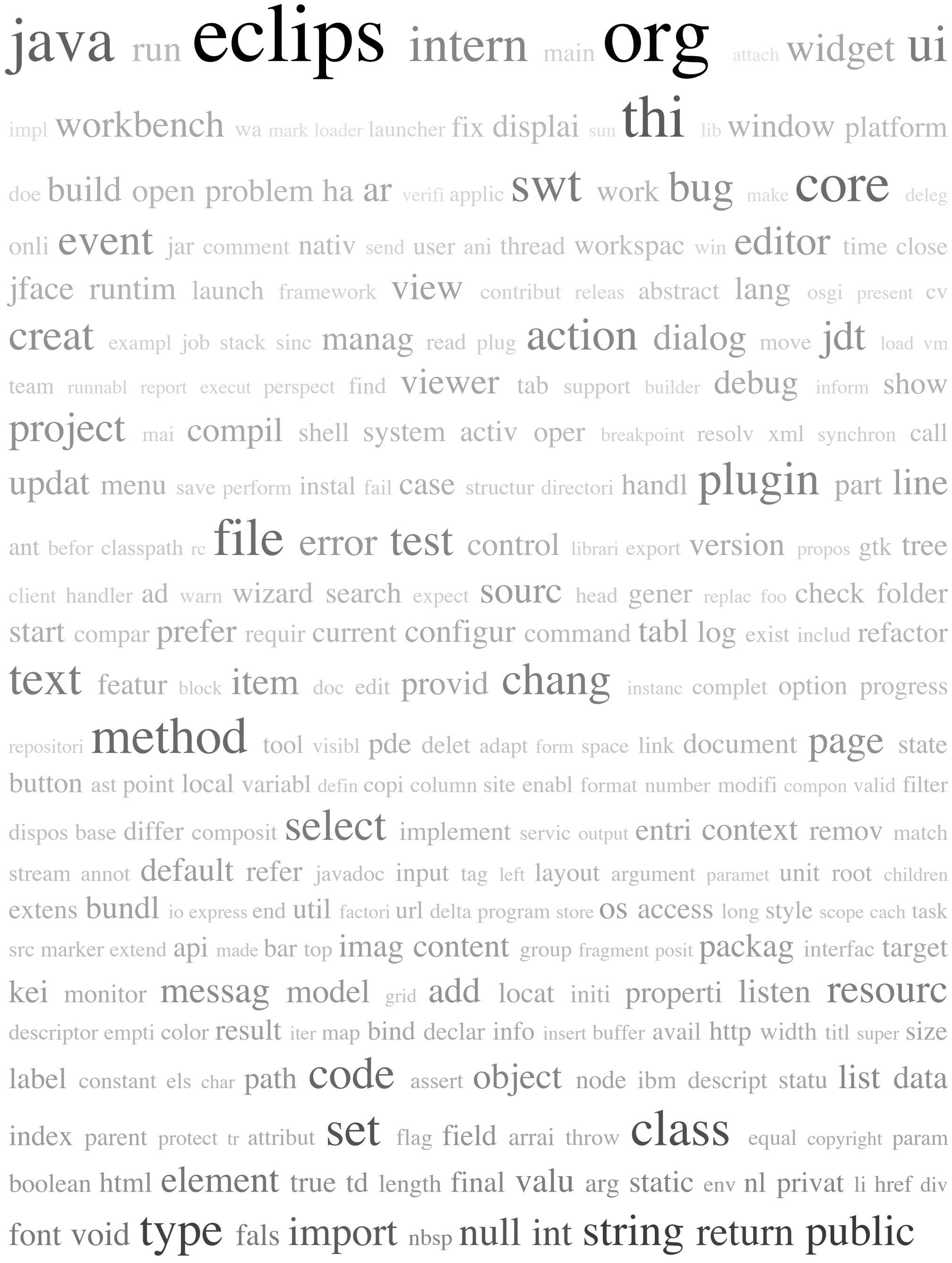

Figure 7.3: Likelihood of vocabulary in bug reports and source code with darkness indicating frequency in source code.

Chapter 8

Conclusion

We presented a novel expertise model of developers. The model is based on the source code vocabulary of developers. The vocabulary of developers is obtained from the `diff` and `log` of their source code contributions. We applied the model in a recommendation system that suggests developers with the appropriate expertise for handling a bug report. The recommendation is based on the lexical similarity between the vocabulary of a developer and the vocabulary of the bug report.

We implemented the approach as a prototype called DEVELECT. We evaluated the recommendation system using two large open source software projects as case studies. We reported on precision and recall for both studies and discussed the results:

Using eight years of Eclipse development as a case study, we achieved 33.6% top-1 precision and 71.0% top-10 recall. When calibrating our approach, we found that developer expertise decays over time. To counter this effect we applied two weighting schemes: i) *decay of vocabulary* weights expertise by a decay factor that is proportional to the time since the developer acquired that expertise, ii) *inactive developer penalty* downgrades developers that had been inactive for a certain time.

Validating these results with a case study on four years of Gnome/Evolution development, we achieved 19.2% top-1 precision and 64.7% top-10 recall. The main challenge that we faced was to establish reasonable links between SVN logins and persons in bug reports. Considering these difficulties and uncertainties, the newly obtained results are decent and confirm the validity of our approach.

It is difficult to precisely and reliably evaluate the performance of our approach. This is due to the choice of the evaluation benchmark and influenced by additional case study-specific problems. These issues mainly affect the evaluation but only to a small degree the approach itself. Therefore, the performance of our approach is probably better than indicated by the evaluation of the case studies.

We briefly investigated the differences and commonalities of the vocabulary used in bug reports and the vocabulary used in source code. The findings showed a reasonable distribution of the different and common words between both vocabularies.

8.1 Future Work

Based on the improvements and extensions of our expertise model and recommendation system already mentioned in chapter 2 and chapter 5, we regard the following as possible future work:

- We would like to extend our expertise model with developer knowledge from other sources, *e.g.* mailing lists, or even combine our approach with approaches that rely on previous bug reports as source (*e.g.* [4, 11, 40, 14]).
- We additionally regard as future work to extend our model with usage expertise as a promising complement to implementation expertise [38].
- Furthermore, we would like to include additional Information Retrieval techniques like Support Vector Machine (SVM), Latent Dirichlet Allocation (LDA) and a weighting by term frequency-inverse document frequency (tf-idf).
- To separately take perfective and corrective contributions into account, we would like to introduce weightings for commit sizes [19].
- Currently our expertise model is limited to the granularity of contributed software changes; a future, more fine grained acquisition of the model could be achieved by using a change-aware development environment that records developer vocabulary as the software is written [32, 37].
- We further consider taking the workload and the preferences of developers into account—the latter using preference elicitation methods—when recommending experts [8].

To additionally improve the performance of our current approach, further potential future work includes:

- Apply an automatic abbreviation expansion when extracting the vocabulary [18]
- Use a copy pattern detection approach to identify source code copying [12]
- Run more fine-grained iterations per bug report and revision
- Improve the matching of login names and bug report-related people

It is difficult to precisely and reliably evaluate the performance of our approach; to simplify the evaluation and at the same time improve the quality of our evaluation benchmark, we would like to perform a future case study where the actual users test and evaluate DEVELECT.

Appendix A

Sample Bug Report

```
<?xml version="1.0" standalone="yes" ?>
<!DOCTYPE bugzilla SYSTEM "https://bugs.eclipse.org/bugs/bugzilla.dtd">

<bugzilla version="3.0.4"
          urlbase="https://bugs.eclipse.org/bugs/"
          maintainer="webmaster@eclipse.org"
>
    <bug>
          <bug_id>249346</bug_id>
          <creation_ts>2008-10-01 12:35 -0400</creation_ts>
          <short_desc>hierarchy view shows duplicates</short_desc>
          <delta_ts>2009-04-28 12:36:31 -0400</delta_ts>
          <reporter_accessible>1</reporter_accessible>
          <cclist_accessible>1</cclist_accessible>
          <classification_id>2</classification_id>
          <classification>Eclipse</classification>
          <product>PDE</product>
          <component>UI</component>
          <version>3.5</version>
          <rep_platform>PC</rep_platform>
          <op_sys>Windows XP</op_sys>
          <bug_status>NEW</bug_status>
          <priority>P3</priority>
          <bug_severity>major</bug_severity>
          <target_milestone>---</target_milestone>
          <everconfirmed>1</everconfirmed>
          <reporter name="Boris Bokowski">Boris_Bokowski@ca.ibm.com</reporter>
          <assigned_to name="PDE-UI-Inbox">pde-ui-inbox@eclipse.org</assigned_to>
          <cc>cwindatt@ca.ibm.com</cc>
          <cc>daniel_megert@ch.ibm.com</cc>
          <cc>Darin_Wright@ca.ibm.com</cc>
          <cc>jerome_lanneluc@fr.ibm.com</cc>
          <cc>remy.suen@gmail.com</cc>
          <cc>zx@eclipsesource.com</cc>
          <long_desc isprivate="0">
            <who name="Boris Bokowski">Boris_Bokowski@ca.ibm.com</who>
            <bug_when>2008-10-01 12:35:38 -0400</bug_when>
            <thetext>Created an attachment (id=114015)
screenshot

I20080930-0921
```

```
See attached screenshot. Should I try to find reproducible steps? It seems to be
     something introduced with the latest I build.</thetext>
          </long_desc>
          <long_desc isprivate="0">
            <who name="Boris Bokowski">Boris_Bokowski@ca.ibm.com</who>
            <bug_when>2008-10-01 12:39:55 -0400</bug_when>
            <thetext>Should this be moved to PDE?  I noticed that the duplicates are
                 really different classes (one from SWT in my workspace, the other from &
                 quot;External Plug-in Libraries&quot;). I can see how I would end up
                 with two classes with the same name, but I don&apos;t think it is valid
                 that they show up in the Hierarchy view under the *same* superclass.
                 There should be two distint superclasses...</thetext>
          </long_desc>
          <long_desc isprivate="0">
            <who name="Dani Megert">daniel_megert@ch.ibm.com</who>
            <bug_when>2008-10-01 12:46:19 -0400</bug_when>
            <thetext>&gt;Should this be moved to PDE?
Yes. Done.</thetext>
          </long_desc>
          <long_desc isprivate="0">
            <who name="Remy Chi Jian Suen">remy.suen@gmail.com</who>
            <bug_when>2008-10-02 03:08:49 -0400</bug_when>
            <thetext>I have this problem as well. I have Widget from gtk sparc then below
                  it I have Control from gtk sparc and also Control from carbon macosx.</
                 thetext>
          </long_desc>
          <long_desc isprivate="0">
            <who name="Darin Wright">Darin_Wright@ca.ibm.com</who>
            <bug_when>2008-10-02 10:06:57 -0400</bug_when>
            <thetext>Boris, do you mean you get duplicates for everything you have in
                 source? since  it&apos;s in the runtime as well, as an external bundle?

I can&apos;t seem to make this happen, but perhaps I don&apos;t quite understand your
     configuration.</thetext>
          </long_desc>

⋮
          <attachment
              isobsolete="0"
              ispatch="0"
              isprivate="0"
          >
            <attachid>121116</attachid>
            <date>2008-12-23 04:40 -0400</date>
            <desc>Workspace to reproduce the defect</desc>
            <filename>workspace.249346.zip</filename>
            <type>application/zip</type>
            <size>1135835</size>
            <data encoding="base64">
                 UEsDBAoAAAAAAGdvlzkAAAAAAAAAAAAAAAARAAAAd29ya3NwYWNlLjI0OTM0Ni9QSwMECgAAAAAA

Z2+XOQAAAAAAAAAAAAAAABsAAAB3b3Jrc3BhY2UuMjQ5MzQ2Ly5tZXRhZGF0YS9QSwMECgAAAAAA

⋮
a3VyL3NpbXBsZXBsdWdpbi9BY3RpdmF0b3IuamF2YVBLBQYAAAAAogKiAmF7AQBk2Q8AAAA=
</data>
          </attachment>
    </bug>
</bugzilla>
```

Listing A.1: A Bugzilla bug report.

List of Figures

Listings

List of Tables

Bibliography

[1] Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from CVS. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 125–128, New York, NY, USA, 2008. ACM.

[2] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing cloning evolution in the Linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.

[3] John Anvik. Automating bug report assignment. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 937–940, New York, NY, USA, 2006. ACM.

[4] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 2006 ACM Conference on Software Engineering*, 2006.

[5] John Anvik and Gail C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[7] Pierre F. Baldi, Cristina V. Lopes, Erik J. Linstead, and Sushil K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 543–562, New York, NY, USA, 2008. ACM.

[8] Olga Baysal, Michael W. Godfrey, and Robin Cohen. A bug you like: A framework for automated assignment of bugs. In *ICPC '09: Proceedings of the 17th International Conference on Program Comprehension*, pages 297–298. IEEE Computer Society, May 2009.

[9] Olga Baysal and Andrew J. Malton. Correlating social interactions to release history during software evolution. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Extracting structural information from bug reports. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 27–30, New York, NY, USA, 2008. ACM.

[11] Gerardo Canfora and Luigi Cerulo. How software repositories can help in resolving a new change request. In *Proceedings of the Workshop on Empirical Studies in Reverse Engineering*, September 2005.

[12] Hung-Fu Chang and Audris Mockus. Evaluation of source code copy detection methods on freebsd. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 61–66, New York, NY, USA, 2008. ACM.

[13] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[14] Giuseppe A. Di Lucca, Massimiliano Di Penta, and Sara Gradara. An approach to classify software maintenance requests. In *Processings of 18th IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 93–102, 2002.

[15] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 341–350, New York, NY, USA, 2007. ACM.

[16] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[17] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, New York, NY, USA, 2008. ACM.

[18] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 79–88, New York, NY, USA, 2008. ACM.

[19] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 Intl. working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

[20] Huzefa Kagdi and Denys Poshyvanyk. Who can help me with this change request? In *ICPC '09: Proceedings of the 17th International Conference on Program Comprehension*, pages 273–277. IEEE Computer Society, May 2009.

[21] Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 184–193, 2004.

[22] Adrian Kuhn. Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 175–178. IEEE, 2009.

[23] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007.

[24] Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 209–218, Los Alamitos CA, October 2008. IEEE Computer Society Press.

[25] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining Eclipse developer contributions via author-topic models. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 30, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, November 2001.

[27] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[28] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140. IEEE, 2009.

[29] Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 5, Washington, DC, USA, 2007. IEEE Computer Society.

[30] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM.

[31] Audris Mockus and David Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2), April 2000.

[32] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 31–34, New York, NY, USA, 2008. ACM.

[33] David S. Pattison, Christian A. Bird, and Premkumar T. Devanbu. Talk and work: a preliminary report. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 113–116, New York, NY, USA, 2008. ACM.

[34] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[36] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, February 2009.

[37] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering)*, pages 317–326, 2008.

[38] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124, New York, NY, USA, 2008. ACM.

[39] Harvey Siy, Parvathi Chundi, and Mahadevan Subramaniam. Summarizing developer work history using time series segmentation: challenge report. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 137–140, New York, NY, USA, 2008. ACM.

[40] Davor Čubranić and Gail C. Murphy. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.

[41] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, Washington, DC, USA, 2007. IEEE Computer Society.

[42] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.