Delft University of Technology
Software Engineering Research Group
Technical Report Series

Identifying Cross-Cutting Concerns Using Software Repository Mining

Frank Mulder, Andy Zaidman

Report TUD-SERG-2010-032

TUD-SERG-2010-032

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Paper accepted at the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution (IWPSE-EVOL 2010).

Identifying Cross-Cutting Concerns Using Software Repository Mining

Frank Mulder
Software Engineering Research Group
Delft University of Technology
The Netherlands
fmuldr@gmail.com

Andy Zaidman
Software Engineering Research Group
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

ABSTRACT

Cross-cutting concerns are pieces of functionality that have not been captured into a separate module, thereby hindering program comprehension and maintainability. Solving these problems requires first identifying these cross-cutting concerns in pieces of software. Several methods for identification have been proposed but the option of using software repository mining has largely been left unexplored. That technique can uncover relationships between modules that may not be present in the source code and thereby provide a different perspective on the cross-cutting concerns in a software system. We perform software repository mining on the repositories of two software systems for which the cross-cutting concerns are known: JHotDraw and Tomcat. Based on the results of the evaluation, we make some suggestions for future directions in the area of identifying crosscutting concerns using software repository mining*.

1. INTRODUCTION

In software development, programmers try to achieve a separation of concerns: each piece of functionality should be implemented in its own distinct module. Object-oriented programming facilitates this separation by providing a system of classes, but research has shown that even when design principles are consciously applied, some concerns do not fit in the existing modularization. These so-called *cross-cutting concerns* [17] lead to two problems: (1) they hinder program comprehension because programmers have to keep track of various concerns while inspecting a piece of code and (2) they decrease maintainability of software since modifying one piece of functionality requires changing code in many places [12, 24]. Identifying cross-cutting concerns is a first step towards solving these problems.

Many methods have been proposed for finding cross-cutting concerns in software systems. Most of these involve finding patterns in source code, while others use dynamic analysis. Another option, which has largely been left unexplored, is to use software repository mining for this purpose.

Software repository mining deals with extracting implicit information from software repositories, e.g. version control systems (VCS) such as CVS and Subversion. Version control systems are commonly used in software development to facilitate working in teams. While not originally designed for mining purposes, we can exploit the (implicit) information stored in those repositories to support software maintenance. In particular, the fact that certain files are often changed together may provide a clue to where cross-cutting concerns are present in the system. This is because developers have to change many entities if the elements of a concern are scattered throughout a system. This idea does rely on the notion of self-contained commits, i.e. transactions in which all files are related to one particular concern. While doing so is generally accepted as good practice and many developers do so, not all projects have such an official commit policy[1]. This should be taken into account when mining version control systems.

Mining version control systems allows to uncover relations that may not be present in the source code, thereby providing a new perspective on finding cross-cutting concerns in a software system. In particular, mining a VCS allows to reveal *logical coupling*: "implicit and evolutionary dependencies between the artifacts of a software system which, although potentially not structurally related, evolve together and are therefore linked to each other from an evolutionary point of view" [10]. This co-change information can be extracted using data mining techniques. As we are mining for items that are frequently changed together, it seems natural to use the technique called *frequent itemset mining*, which is able to discover interesting relations in a database, e.g., relations of the type "customers who bought product x also bought product y". This leads us to the central research question of this paper:

> *Can we apply frequent itemset mining on version control system data to find cross-cutting concerns in a software system?*

Answering this question requires developing a tool that outputs frequent itemsets for a given version control system. Those itemsets are cross-cutting concern candidates

*This work is described in more detail in the MSc thesis of Frank Mulder [25].

[1]The KDE project prescribes the following in its SVN commit policy: "Please commit all related changes in multiple files [...] in the same commit" and "Every bugfix, feature, refactoring or reformatting should go into an own commit"; see `http://techbase.kde.org/Policies/SVN_Commit_Policy#Commit_complete_changesets`

IWPSE-EVOL'10 September 20-21, 2010 Antwerp, Belgium

and should be checked to see if they actually represent cross-cutting concerns. A short literature survey that we performed revealed that *evaluation* of aspect mining techniques is a weak point of many studies; a conclusion shared by Mens et al. [22]. Furthermore, many evaluations are of a subjective nature and there is a lack of quantitative information, which makes comparing techniques difficult. Therefore, we will pay particular attention to the evaluation of our results: manual assessment is avoided in favour of automatic evaluation against known cross-cutting concerns in benchmark systems.

Mining a version history can be done on various levels of granularity: we can consider the names of the files which have been changed in each transaction, or we can mine at the finer-grained level of methods. Our research takes both file-level and method-level mining into account and we will describe how well both techniques perform in terms of speed and result accuracy.

This paper is structured as follows: we start with an overview of previous research related to the topic of this paper in Section 2. Next, the design and implementation of our tool-chain are described in Section 3. Section 4 then discusses the results of applying our tool-chain on two different software systems. Finally, in Section 5 we draw conclusions from these results and suggest directions for future work.

2. RELATED WORK

The act of finding cross-cutting concerns is often called "aspect mining". A detailed survey of aspect mining techniques was performed by Kellens et al. [16] and we list some notable techniques in this section.

Identifier analysis is based on the idea that cross-cutting concerns are often implemented by the rigorous use of naming and coding conventions [23] and groups together identifiers that have similar names or meanings. *Fan-in analysis* identifies methods that are called from many different places as cross-cutting concern candidates [21]. *Clone detection* exploits the fact that when it is not possible to restrict a concern to one module, developers are forced to write the same code over and over again [6]. *Dynamic analysis* uses program traces to find recurring execution patterns, thereby identifying cross-cutting concern candidates [3, 26]. Finally, *history-based aspect mining* uses information from version control systems to find cross-cutting concern candidates.

Not much research has yet been done on history-based aspect mining. Actually, only one research group has done a study in which they actually found cross-cutting concerns using a software repository. That group consists of Breu, Zimmermann and Lindig; they coined the term HAM: History-based Aspect Mining [4, 5]. Similar to fan-in analysis, they consider the number of calls to certain methods (i.e. the fan-in). In contrast to fan-in analysis, which considers the fan-in values of the methods in a certain snapshot of the source code, HAM looks at additions of methods calls (i.e. increases in fan-in). It is unclear whether this historical perspective adds anything to "plain" fan-in analysis. In fact, HAM fails to exploit the logical coupling information that a software repository provides. The technique we describe in this paper does use this information, by considering frequent additions and modifications of files and methods.

Another relevant study has been done by Canfora, Cerulo and Di Penta, in which the evolution of cross-cutting concerns in the JHotDraw application was investigated [8]. Their technique relies on a known set of concerns and thus cannot identify them on its own. However, their research provides some nice insights into how cross-cutting concerns evolve over time. It appears that cross-cutting concerns are often introduced in one transaction and then extended in later transactions (an observation also made by Breu et al. [4]).

A more recent contribution to the body of knowledge in this area comes from Adams et al. [1]. They present COMMIT, a history-based cross-cutting concern mining approach which uses a robust, statistical clustering mechanism to deal with small and even large variations in the instances of a concern throughout time. They applied COMMIT on PostgreSQL and NetBSD and compare their results to CBFA (Clustering-Based Fan-In analysis) [29] and HAM [4, 5] and obtained good results. They also show that COMMIT is complementary to CBFA and HAM.

Research in the area of software repository mining has been surveyed by Kagdi et al. in 2007 [15].

3. TOOL-CHAIN STRUCTURE AND IMPLEMENTATION

This section describes the tool-chain with which we identify cross-cutting concern candidates. As indicated in the introduction, the tool should be able to mine frequent itemsets from a version control system. It should also have the ability to handle large systems with long histories and to evaluate the resulting itemsets in a systematic way.

First, our technique is described in terms of a common framework. Next, we discuss the various modules in our tool-chain, including design decisions and implementation details.

3.1 Fitting in a Common Framework

Marin et al. [20] propose a common framework for aspect mining, which "allows for consistent assessment, comparison and combination of aspect mining techniques". Their framework requires various parts of a technique to be defined in a consistent way. First of all, the *search goal* defines what kinds of cross-cutting concerns the technique aims to identify; a classification of 13 cross-cutting concern sorts[2] to choose from has been made by the same authors [19]. Their framework also prescribes that the format in which the results of the aspect mining process are *presented* should be defined. Furthermore, we should define the relation between the mining results and the targeted concerns; this *mapping* also describes how we should understand and reason about those results. Finally, we should define the *metrics* to assess the mining technique's results. Our technique can be explained in the terms of that framework with the following definitions:

Search goal Finding those concerns that exhibit frequent co-change behaviour. One may think of concern sorts such as Consistent Behaviour and Contract Enforcement, but these will only be identified when the method implementing the desired functionality is renamed (please see Marin's concern sort classification [19] for an explanation of these sorts). Concerns of the Expose Context and Exception Propagation sorts may very well be identified by our technique, as both require changes of every method in a call stack.

[2] *Sorts* are generic descriptions of crosscutting functionality that can ease classification of cross-cutting concerns

Presentation Itemsets, consisting of the names of entities (files or methods) that were frequently changed simultaneously.

Mapping The entities in the itemsets principally match the cross-cut elements but entities implementing cross-cutting functionality may show up in there as well.

Metrics Starting from a known set of cross-cutting concerns for the subjects that we analyse, we use the so-called F1 measure to determine how well an itemset represents a cross-cutting concern (see Section 3.5.2 for details). For each itemset we determine the best matching concern, i.e. the one with the maximal F1.

3.2 Tool-chain Structure

Our tool-chain follows the typical steps of software repository mining tools: data acquisition, processing and presentation. More specifically, it acquires data from a Subversion repository and transforms them into changesets (which consist of the names of entities added or modified in each transaction). These are then processed using frequent itemset mining and the resulting itemsets are analysed to get cross-cutting concern candidates. Finally, the user is presented with a list of itemsets, which can also be used to produce graphs for visualising the results.

Our tool-chain is currently limited to analysing systems written in Java but can easily be adapted to other object-oriented languages.

Our approach is able to mine the version control system for evidence of co-change at two levels of granularity, namely file-level and method-level. We now discuss the pros and cons of both:

File-level mining This level of mining deals with files that were frequently changed together. Advantages are that it requires little effort to extract the data, and the number of entities to analyse will be relatively small (compared to method-level mining), with short execution times as a result. Additionally, non-source code files (e.g. configuration files) can also be identified as being part of a concern. A disadvantage is formed by the fact that file-level mining is not very precise (probably leading to false positives in the result set).

Method-level mining On this level, a syntactic analysis of the source files is performed, such that additions and modifications of methods can be recorded. An obvious disadvantage is that we need to download the contents of files to analyse their contents. The syntactic analysis that is needed to find out which methods have been modified also takes time. As files typically contain more than one method, the resulting data set will be larger than the one we got at file-level mining. The data mining algorithm that analyses these data will therefore take longer to execute and will also have larger memory requirements. However, the results of method-level mining are probably more precise than those found with file-level mining, making them more useful as cross-cutting concern candidates.

We clearly have to make a trade-off between the time and memory requirements of the technique and the result accuracy. We will shed more light on this when discussing the results of our case studies. But first, the next sections discuss each module of the tool-chain in more detail.

3.3 Data Acquisition

As indicated in the previous section, our tool-chain operates on Subversion repositories. As CVS repositories can be converted to a Subversion repository using `cvs2svn`[3], we are also able to deal with CVS repositories. We use SVNKit[4], a Java Subversion library, to fetch entries from a Subversion repository. Similar to what Breu et al. [4] did, we reinforce (combine) two changesets if they come from transactions that have been committed by the same author within a certain period of time. This compensates for the behaviour of some programmers to frequently commit small transactions which are actually related.

Some of the changesets are filtered to avoid noise: for example, a file may be committed both to a branch and to the trunk, making it appear twice in the output. Filtering may also help in avoiding irrelevant data at an early stage: we can filter out changesets containing very few or very many items if they produce itemsets that are not likely to be valid cross-cutting concern candidates. In addition, we keep track of files that have been renamed, moved or copied using Subversion's `copy` command.

As indicated before, method-level mining additionally requires a syntactic analysis of the source code files. This is done using a modified version of DiffJ[5], which is like the Unix program `diff`, but specifically for Java code. The unmodified version only reports changes in the highest node in an abstract syntax tree. We modified the source code in order to capture every method addition, even when this is done as part of the addition of a class. We also modified it in such a way that it outputs the changes in a format usable by our tool-chain.

3.4 Frequent Itemset Mining

A common way to mine patterns in a database is *frequent itemset mining* (FIM). Informally, this means that we search for sets of items of which the number of occurrences is above a certain threshold. Frequent itemsets are typically used as the first step in association rule mining. Many people are familiar with association rule mining in the context of online stores, where recommendations such as "customers who bought product x also bought product y" are given. For example, the frequent itemset $\{Bread, PeanutButter\}$ may have been found (meaning that these products were often bought together), and from this, the rule $Bread \Rightarrow PeanutButter$ is generated, generalising the previous statement by not only noting that they are often bought together, but also concluding that there is a relation between these two.

For our purpose, we only need the frequent itemset mining part: we are just looking for sets of entities that are commonly changed together, and we do not need to generate rules, although we do assume that the frequent occurrence of sets of entities implies that there is a relation between those entities [30]. What follows is a formal definition of frequent itemsets.

3.4.1 Definition

Let $I = \{I_1, I_2, \ldots, I_m\}$ be a set of items and $X \subseteq I$ an itemset. Further, define database D as a set of transactions:

[3] `http://cvs2svn.tigris.org`
[4] `http://svnkit.com`
[5] `http://www.incava.org/projects/java/diffj/`

$D = \{t_1, t_2, \ldots, t_n\}$, where $t_i = \{I_{i1}, I_{i2}, \ldots, I_{ik}\}$ and $I_{ij} \in I$. Also, let $t(X)$ be the set of transactions that contain itemset X, formally $t(X) = \{Y \in D | Y \supseteq X\}$. Finally, the *support* of an itemset X is the fraction of transactions in the database that contain X: $support(X) = \frac{|t(X)|}{|D|}$ [2, 11].

Then X is called a *frequent itemset* when its support is higher than a given minimum support: $support(X) \geq minsupport$. The set of all frequent itemsets is denoted by **FI**; it is a subset of the power set of I, i.e. $\mathbf{FI} \subseteq 2^I$.

3.4.2 Algorithm requirements

In association rule mining, the most relevant itemsets are those with a large support and confidence. Support was defined in the previous section and confidence is defined as follows: $conf(X \Rightarrow Y) = supp(X \cup Y)/supp(X)$. In association rule mining these values are used to determine whether an association rule is valid or not. This means that if a frequent itemset mining algorithm is used as part of association rule mining, its running time can be reduced by setting a high minimum support or confidence. However, confidence cannot be used in our case, as frequent itemsets do not have a direction (whereas association rules do have that: $X \Rightarrow Y$ is not the same as $Y \Rightarrow X$; hence the corresponding confidence values will be different). And even while a higher support also means a more relevant itemset in association rule mining, this may not be the case for our technique. The cardinality of itemsets could be at least as important when identifying cross-cutting concerns, as changing one concern can lead to changes in many files.

Therefore, we would like to analyse itemsets that do not occur very frequently as well (even with a number of occurrences as low as 2). This means that an algorithm should be able to complete within a reasonable amount of time even when run with a low minimum support.

Also, it should be able to deal with the characteristics of change history data. In a study of the nature of commits in various software systems, it appeared that 80% of the commits were tiny (which in this case means that less than 5 files were changed in these commits) [14]. However, there were also commits in which a relatively huge number of files were changed. Thus, we notice a mix of dense and sparse input data. In the most well-known FIM algorithm, Apriori, the potential number of database scans is 2^m [11], where m is the size of the largest transaction in the database, so the occasional dense data tremendously hampers the performance of this algorithm.

Before considering other FIM algorithms, we may wonder: do we actually need all the output we got from performing frequent itemset mining? For our purpose, we are not interested in itemsets that are subsets of itemsets that have the same support. It appears that generating only these relevant itemsets can be done a lot faster than generating all frequent itemsets. The itemsets we are looking for are called *frequent closed itemsets*, and are discussed next.

3.4.3 Frequent Closed Itemsets

A frequent itemset is called *closed* when no supersets with the same support exist (i.e. if its support is different from the supports of its supersets) [13]. Formally, an itemset X is closed if it satisfies $I(t(X)) = X$, where $I(\mathcal{S}) = \bigcap_{T \in \mathcal{S}} T$, $\mathcal{S} \subseteq D$ (recall that $t(X)$ means the transactions that contain X, and D is the set of all transactions) [27]. Call the set of frequent closed itemsets **FCI**, then it holds that $\mathbf{FCI} \subseteq \mathbf{FI}$ (in practice, $|\mathbf{FCI}|$ is orders of magnitude smaller than $|\mathbf{FI}|$) [7].

We tested the performance of several open source frequent closed itemset mining implementations, by running them on 1523 changesets of the repository of ArgoUML (on file-level), with the minimum number of occurrences set to 2. While Apriori, the classical FIM algorithm could not complete, even with a much higher minimum support, the fastest Linear time Closed itemset Miner (LCM) implementation completed in 3 seconds.

In contrast to other algorithms, which basically enumerate frequent itemsets and then prune away unnecessary sets, LCM only generates frequent closed itemsets. This means that the algorithm is linear in the number of frequent closed itemsets. Also, several techniques are used to speed up computation, in particular a technique which adjusts to parts of the input being dense or sparse.

Apart from frequent closed itemset mining, one might also consider *maximal* frequent itemset mining. The set of maximal frequent itemsets is orders of magnitude smaller than the set of frequent closed itemsets and can be generated much faster. A set is called maximally frequent if it has no frequent supersets (in contrast to closed sets, no restriction is imposed on the support of those supersets). As we said in the previous section, we are also interested in itemsets with a large cardinality, even when their support is lower, so maximal FIM does not do what we want. Thus, frequent closed itemset mining appears to be the right choice.

3.5 Itemset Analysis

The goal of itemset analysis is to mark certain itemsets (from the previous step) as cross-cutting concern candidates. To this end, we need to specify parameters based on which we can select those itemsets, and we should evaluate to what degree a candidate represents a cross-cutting concern.

3.5.1 Selecting Cross-Cutting Concern Candidates

We have got the following criteria and parameters at our disposal for selecting itemsets as cross-cutting concern candidates:

Support Itemsets that occur frequently may be more relevant; if so, we can discard itemsets with a low support.

Cardinality Itemsets containing very few or very many items may be irrelevant.

Lift Lift divides the actual support by the support that would be expected by chance (i.e. if the files were committed independently): $lift(X \cup Y) = supp(X \cup Y)/(supp(X) \cdot supp(Y))$. This might give a more accurate impression of how relevant an itemset is.

Changeset size We can discard very small and/or very large changesets to limit the input of the mining module, to limit the production of irrelevant itemsets.

Reinforcement interval The reinforcement interval influences how many changesets are combined and consequently influences the resulting frequent itemsets.

To determine the constraints on the above criteria to get as many relevant itemsets as possible (compared to the total collection of itemsets found), we first establish what makes an itemset relevant; this is what the next section is about.

3.5.2 Evaluation Criteria

In order to find out which candidates actually represent cross-cutting concerns, we evaluate against known sets of

cross-cutting concerns. Those sets describe for each concern which methods take part in it. For file-level mining, we discard the method information, keeping only the names of the files belonging to each concern. Using those data, we can determine the precision and the recall for each itemset. Maximising either one does not really make sense; for example, we could always achieve a high precision by selecting sets with very few items (as all items in the set will then belong to a concern), but recall would be very low. Therefore, we use a combination of precision and recall: the F1 measure, which is commonly used in information retrieval and which is the harmonic mean of precision and recall [18]:

$$\begin{aligned} precision &= \frac{|\text{Relevant} \cap \text{Retrieved}|}{|\text{Retrieved}|} \\ recall &= \frac{|\text{Relevant} \cap \text{Retrieved}|}{|\text{Relevant}|} \\ F_1 &= \frac{2 \cdot (precision \cdot recall)}{(precision + recall)} \end{aligned}$$

Here 'Relevant' is the set of those items that appear in the cross-cutting concern, and 'Retrieved' means those items that appear in the itemset. The F1 score is a value between 0 and 1, where 1 is the best score.

For each itemset we calculate the maximal F1, i.e. the score for the cross-cutting concern that best matches this itemset. This value then determines how relevant an itemset is. It makes sense to use this value because this is also how one would evaluate an itemset manually: one would try and see which concern it matches best and then determine how well it matches that concern. To get an overall score for the itemsets, we take the average of these values: the average maximal F1. This score can be seen as an alternative precision measure for the set of itemsets.

3.6 Presentation

The presentation of the results is pretty straightforward: a comma-separated value file is generated, which contains the itemsets along with the transactions in which they occur. This file can also be used to generate graphs to get various views on the itemsets.

4. EMPIRICAL STUDY

This section describes the results of running our tool-chain on two different case studies.

4.1 Choice of Subjects

Finding good subjects for testing our tool-chain is quite a challenge: as we decided to do an automated evaluation, there needs to be a known list of cross-cutting concerns for the systems we are going to analyse. Developers usually do not document the concerns in their system, so we have to rely on results from existing aspect mining research, which are not always publicly available. Marin et al. [21] have put effort into "setting up a web forum[6] where aspect mining researchers can exchange and discuss aspect candidates found in (open source) software systems". Unfortunately, only the results from their own research (mainly from fan-in analysis) are available there. Still, it seems logical to use the results from that web site for evaluating our own technique, as it is the only one on which a systematic overview of the concerns in various software systems is given. Especially the results from JHotDraw, a program that is frequently used as a benchmark in aspect mining research, are of interest. On the same web site, the results for two other subjects are given: Tomcat and PetStore. We considered PetStore to be too small to get a relevant evaluation (only 7 concerns are listed) but Tomcat does seem relevant as a test subject as it is a 'real' application (as opposed to JHotDraw, which is more of a design exercise) with a lot of concerns.

Therefore, we have chosen to run our technique on the repositories of JHotDraw and Tomcat. The next sections will discuss the results for each application in detail.

4.2 Case Study: JHotDraw

4.2.1 Application Overview

JHotDraw is a Java GUI framework for technical and structural graphics[7]. It has been developed as a design exercise, showing good use of design patterns. For this reason, JHotDraw has frequently been used in aspect mining research: if some part of the system has not been separated in a module but is somehow scattered over the system or tangled with other code, we can safely assume that this is not because of sloppy programming and that it forms a valid cross-cutting concern. On the other hand, this makes it a somewhat "artificial" subject. JHotDraw's history started on 8 August 2002 and is currently at revision 544. 11 different developers have committed transactions and the application is still being actively developed.

4.2.2 Evaluation Set

Canfora et al. [8] kindly provided us with the list of cross-cutting concerns for JHotDraw, which, though identical to the list given on the aforementioned web site, did not only contain the names of the method implementing the cross-cutting behaviour but also the names of the cross-cut elements. As we want to know the complete scope of a concern, this is very useful for us.

Because JHotDraw is often mentioned as a benchmark in aspect mining literature, we started searching for publicly available aspect mining results for JHotDraw. The results for the dynamic analysis experiment with the Dynamo tool by Tonella and Ceccato [26] were publicly available[8]. Ceccato et al. also mention identifier analysis as one of the techniques tested on JHotDraw [9] and they kindly provided us with the results of this experiment. Zhang's PRISM Aspect Miner results [28] are publicly available[9] but they were not usable as the actual cross-cutting concerns were not reported (only separate entities that may be part of a concern).

Although we would want to have one list of cross-cutting concerns in JHotDraw, with the results of the various experiments merged together, such a list does not exist as yet. Therefore, some concerns that actually belong together are reported as separate concerns (for example, all three result sets contain a concern called "Undo"). In total, we have a list of 72 concerns for JHotDraw, available in [25].

The studies from which we use the results were all done on JHotDraw version 5.4b1. In the meantime, JHotDraw has progressed to version 7.0.8 and the package structure has changed, so we decided to only mine JHotDraw's history

[6] http://swerl.tudelft.nl/amr/

[7] http://www.jhotdraw.org

[8] http://star.itc.it/dynamo/jhotdraw-detailed-results.html

[9] http://www.eecg.utoronto.ca/~czhang/mining/j6.txt

until version 6, which has the same package structure as version 5.4b1. This part consists of 172 revisions.

4.2.3 Results for File-Level Mining

When running our tool on the repository of JHotDraw while considering file names, we get 828 itemsets, with an average maximal F1 of 0.36 (running time: 3 seconds[10]). Subsequently, we tried to improve this score by tweaking the reinforcement interval and the minimum and maximum values for support, cardinality, lift and changeset size: 9 parameters in total. As these parameters are probably not independent, it would be best to test all possible combinations of all possible values for each constraint. This seems not feasible, however, as we are dealing with 9 parameters, each of which can have many different values. We will therefore change these parameters separately and try to get good outcomes that way.

We start with the reinforcement interval, as it influences all other criteria: combining changesets will lead to different changeset sizes and different itemsets. We ran our tool with reinforcement intervals between 0 and 10000 seconds, with steps of 100 seconds. An interval of 1700 seconds yielded 682 reinforced changesets and gave the best result: the average maximal F1 became 0.40, which is a bit more than without reinforcement. The results we got with reinforcement were always better than without, so it seems indeed worthwhile to combine changesets if they have been committed close in time by the same author.

With the reinforcement interval fixed to 1700, we started investigating the effects of putting constraints on the changeset size. We set the minimum changeset size between 0 and 40 and the maximum size between 40 and 500, and tried all different combinations (with steps of 1 for the minimum and steps of 10 for the maximum because there are fewer large changesets than small ones). By constraining the changeset size between 9 and 40 we reached the highest score: 0.53. This would suggest that discarding both the smallest and the largest changesets improves the final score. However, it is interesting to see that setting 40 as the minimum or maximum changeset size both give the same result. This suggests that the size of a changeset is not very much related to the question whether it contributes to a concern. Another thing is that constraining the changeset size too much will leave very few itemsets. For example, the aforementioned constraints of 9 and 40 lead to 11 changesets, eventually resulting in 18 itemsets. Although having few itemsets could be a good thing (as it will take less time for a user to analyse them) it can also mean that much relevant information has been discarded. Not only do we risk throwing away relevant itemsets, the itemsets that do appear in the outcome are significantly smaller, consisting of only a few items. As we want to cover as many elements of a concern as possible, this is not a desirable result. To make sure we do not lose any useful data, we continue our parameter investigation with the changeset size unconstrained.

For investigating the relation between the number of occurrences and cardinality of an itemset and the resulting maximal F1 values, please take a look at Figure 1. That graph shows for each number of occurrences and size the maximal F1 score for the itemset that best matched a concern (we have chosen to limit the "cardinality" axis to only

[10]All experiments were run on dual quad core Intel Xeon E5345 2.33 GHz processors with 16GB of RAM.

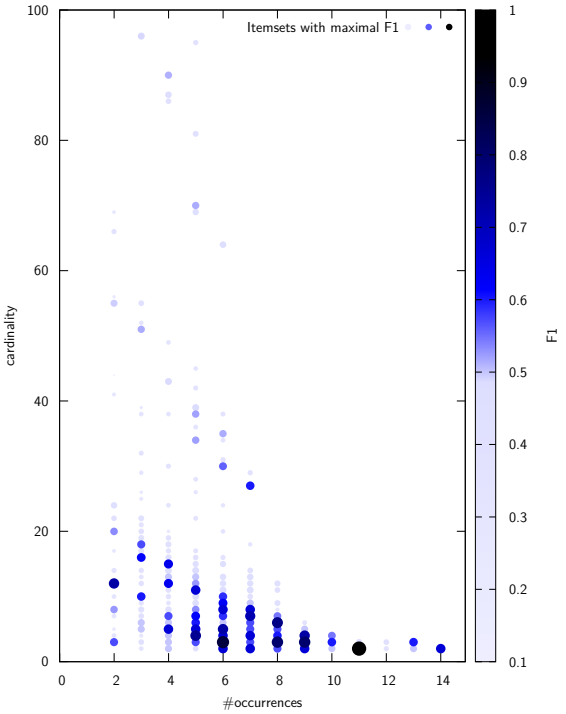

Figure 1: Itemsets with maximal F1 for the given cardinality and number of occurrences (JHotDraw, file-level). Colour intensity and size represent the F1. Itemsets with F1 $\geq$ 0.5 are annotated with the corresponding concern identifier.

show values under 100 as itemsets with a larger cardinality have low F1 scores and showing more values would make the graph unreadable; the same holds for the "#occurrences" axis). The F1 is represented by both the colour intensity and the size of the dot. An extended version of this graph, in which the itemsets are annotated with an abbreviation of the corresponding best-matching concern, can be found in [25], along with the complete list of concern abbreviations and their meanings. We notice the following things in Figure 1:

1. No itemsets that occur less than 2 times or that consist of less than 2 items appear.
2. Itemsets with a large cardinality are relatively less frequent than those with a small cardinality; itemsets that occur frequently usually have a small cardinality.
3. "Sweeps" of similar concerns occur in various places[11]. For example, the same concern can be found at (#occurrences, cardinality) coordinates (2,20), (3,18), (3,16), (4,15), etc.
4. "Relevant" concerns are not concentrated in one area.

Especially point 4 is relevant for our research, but let us first discuss the other three observations. The first point is

[11]In the original version of Figure 1 each dot is accompanied by an identifier which indicates the concern. This makes it possible to see the sweeps. We removed the identifiers in the paper in order not to clutter the image.

related to the parameters of the mining algorithm: itemsets with less than 2 items are not relevant because we want to see files that have been committed together.

The second point is not surprising, albeit a bit disappointing. If we had found large groups of files that were changed together very frequently, those itemsets would be really interesting. In practice, however, commits consisting of large files are apparently not frequent, and groups of files that are frequently committed together are usually small (a similar observation was made by Hattori and Lanza [14]).

To understand the third point, we looked at (1) the number of occurrences of the changesets, (2) the cardinality of the itemsets, (3) the best-matching concern for each itemset and the corresponding F1 value and (4) in which changesets the items in the itemsets occurred (see Table 1). From these data, we observed that the DOD (Manage figures outside drawing) concern is matched by multiple itemsets and that these itemsets have a common core of items in them. Also, the transactions where the items in these sets occur are similar. The reason for this is that we have chosen to do frequent closed itemset mining (as discussed in Section 3.4.3): it will output an itemset even if it has frequent supersets, leading to many similar itemsets. It turns out that it was a good decision not to use maximal frequent itemset mining, as that technique would not output the best-matching itemset for the DOD concern (because it has frequent supersets).

One might suggest to discard the supersets of an itemset if they have a lower F1 score anyway, but how would one know which itemset is the "best" one of a group of similar itemsets? In this case we could pick the itemset with the highest number of occurrences, but this will not always work. For example, the itemsets at (2,12) and (3,10) in Figure 1 both match the same concern, but the itemset at (2,12) has the highest F1.

This leads us to the fourth point, as we would like to find a relation between the input parameters and the resulting relevance. However, such a relation does not seem to exist. Although most relevant itemsets seem to have a low cardinality, this holds for the itemsets in general. Consequently, restricting the cardinality to low values does not help in improving the score. We tested this again by trying different constraints for the cardinality, and we got the best result for a minimum of 4 and a maximum of 20, leading to a slightly improved final score of 0.42.

Finally, restricting the support can help a little bit, but this gives the same problem as we had when restricting the changeset size: if we want to achieve a higher score than before, we have to restrict the support so much that we throw away a lot of relevant itemsets. For example, we can achieve a score of 0.5 by setting both the minimum and maximum to 14 but this leaves only 9 itemsets.

There is still one parameter left to explore: the 'lift', which is the support divided by the support that would be expected by chance. If some files were committed much more often together than we would expect, they might form a likely cross-cutting concern candidate (in other words, if the lift of an itemset is high, it should be more relevant). However, an investigation of this matter reveals that in fact the opposite is true: the more relevant itemsets (with a high F1) have a relatively low lift.

It follows that using this measure instead of the support is not really an improvement. However, if we could find a better model for the chance that several files are committed together (instead of assuming that all files are independent), we might be able to create a measure that is more suitable for discriminating relevant itemsets. This is something that should still be investigated.

We have tried to achieve a high maximal F1 for the given input: with reinforcement enabled, we reached a score of 0.40. Restricting the other parameters, we could slightly improve this, but we found that many itemsets were discarded this way. We also found that there was no immediate relation between the parameters (in particular support and cardinality) and the final score. This means that we cannot rank the itemsets to put the more relevant ones on top.

Let us take 0.40 as the final score for this particular case study. Now what does this value actually mean? As the score can take on values between 0 and 1, we can say that this value is at the low end of the spectrum. For one itemset, an F1 of 0.4 could for example mean that both the recall and precision are 0.4, meaning that 40% of the items in the matching concern was retrieved and that 40% of the retrieved items was relevant. Someone who takes these results as a starting point for finding cross-cutting concerns would thus have to weed through 60% of irrelevant items, and still only 40% of the concern as we know it would be reported. Although this suggests that the score is relatively low, it does not really say anything until we have some reference point. As other techniques do not report similar values to compare ours with, we decided to perform tests with randomly generated itemsets, to provide a kind of lower bound for the final score. We generated as many itemsets as our tool-chain produced based on the changesets of JHotDraw, with a similar average itemset cardinality and with the same set of entities (files, in this case). From these itemsets we again calculated the average maximal F1. We ran this procedure 1000 times and took the average of all scores. This led to the value of 0.13, which means that the score of 0.40 is about 3 times better than the random case.

4.2.4 Results for Method-Level Mining

Performing method-level analysis of JHotDraw's repository yields 403 itemsets with an average maximal F1 of 0.15 (running time: 45 seconds). We can again try to improve this value by using the parameters we also used when performing file-level mining. Please note that we did not include a graph similar to Figure 1 for reasons of space, but that it is available in [25].

For the reinforcement interval we did a similar test as with file-level mining, but this time the results were quite different: applying reinforcement gave an identical score of 0.15. Moreover, the results with reinforcement were almost always lower than without. This suggests that combining changesets is actually not really a good idea; apparently the changesets are already quite self-contained. The higher score in the file-level case may have been caused by false positives.

Likewise, constraining the changeset size did not help at all, as the scores were always lower than without doing so.

Restricting the itemset cardinality helps somewhat, just like in the case of file-level mining, but the best scores were achieved by setting the minimum cardinality to 20, which again means that some relevant itemsets were discarded.

Setting the minimum number of occurrences to 8 increases the average maximal F1 to 0.18, but it does not make sense to do so as this score is then based on only 1 itemset. Lower choices for this value result in a final score that is no signifi-

#Occurrences	Cardinality	F1	Concern	Changesets	Itemset
11	2	1	DOD	[1, 2, 3, 7, 8, 12, 21, 22, 37, 39, 76]	[standard/CompositeFigure.java, standard/StandardDrawing.java]
8	3	0.8	DOD	[1, 3, 7, 8, 12, 21, 22, 76]	[figures/TextFigure.java, standard/CompositeFigure.java, standard/StandardDrawing.java]
9	3	0.8	DOD	[1, 3, 7, 8, 12, 22, 37, 39, 76]	[samples/javadraw/JavaDrawApp.java, standard/CompositeFigure.java, standard/StandardDrawing.java]
8	3	0.8	DOD	[3, 7, 12, 21, 22, 37, 39, 76]	[contrib/GraphicalCompositeFigure.java, standard/CompositeFigure.java, standard/StandardDrawing.java]
8	6	0.77	IB23	[1, 3, 7, 8, 10, 11, 12, 76]	[applet/DrawApplet.java, application/DrawApplication.java, samples/javadraw/JavaDrawApp.java, standard/CreationTool.java, standard/SelectAreaTracker.java, standard/StandardDrawingView.java]

Table 1: Example itemsets for JHotDraw on file-level

cant improvement over the score we get without a restriction on the number of occurrences.

Experimenting with the lift again showed that this measure does not help us in discriminating relevant itemsets.

Again, we compared the result with the random case, following the same procedure as with file-level mining. The result for the random case was 0.06, which means that the result we got for the changesets of JHotDraw is about 3 times better than that.

4.3 Case Study: Tomcat

4.3.1 Application Overview

Apache Tomcat is an open source software implementation of the Java Servlet and JavaServer Pages technologies[12].

4.3.2 Evaluation Set

For Tomcat we only have one list of concerns, i.e. the results from fan-in analysis, consisting of 48 concerns. That analysis was performed on Tomcat version 5.5, so we decided to mine the part of the Tomcat repository that eventually ended up in the `5.5.x` branches (taking account of copied paths, as recorded by both CVS and Subversion); this part consists of 13490 changesets.

4.3.3 Results for File-Level Mining

Mining Tomcat's repository on file-level resulted in 7261 itemsets with an average maximal F1 of 0.11 (running time: 65 seconds). This score is significantly lower than the score we got for JHotDraw but it should be noted that we only have one source of known cross-cutting concerns for Tomcat. Using only fan-in analysis information for evaluating JHotDraw also leads to a lower score: we get a score of 0.24 as opposed to 0.40 which we got with the complete concern set.

As for reinforcing changesets, we got a similar result to what we got when mining JHotDraw on file-level: the results with reinforcement were always better than without. However, the gains were pretty low: after rounding we still got 0.11 as the final score.

The results for limiting the changeset size were not much different, as the best score we could get by doing this was again 0.11. The same holds for limiting the itemset size.

Setting a high minimum support led to higher F1 values, just like in the previous cases, but again this means that we have to discard many relevant itemsets.

The result for the random case was 0.05, which means that the performance of the tool-chain based on Tomcat's changesets (on file-level) was about 2 times better.

4.3.4 Results for Method-Level Mining

Mining Tomcat's repository on file-level resulted in 5062 itemsets with an average maximal F1 of 0.03 (running time: 11 minutes).

For analysing the results of reinforcing changesets, we decided to test the intervals that gave the highest results in the previous cases because testing as many intervals as we did before would take a very long time. For these intervals we did not get higher F1 scores than without reinforcement.

This time, however, constraining the changeset size did improve the final score significantly: from 0.03 to 0.07, for a minimum of 40 and a maximum of 400. Just like in the file-level mining experiment on JHotDraw, this means that we get less itemsets with less items in it.

Setting the minimum cardinality to 50, we managed to increase the final score up to 0.10. When we combine this with the previously mentioned constraints on the changeset size, which leaves almost only itemsets matching one specific concern (concern B9 to be precise, see [25] for details), we can even get a score of 0.40. However, we should keep in mind that having many itemsets that all match one concern is only of limited use.

The result for the random case was 0.01, which means that the performance of the tool-chain based on Tomcat's changesets with the default parameter values was about 3 times better, and the result with various constraints set was even much better.

[12] `http://tomcat.apache.org/`

4.4 Threats to Validity

Some factors may threaten the validity of our results. First of all, we cannot be 100% sure that the cross-cutting concern information we used for evaluating the results is good enough to be conclusive about the applicability of frequent itemset mining for identifying cross-cutting concerns. It is probably not complete in the sense that it will not contain all cross-cutting concerns, but it is based on earlier research done on cross-cutting concerns in those systems so for the moment it is the best we can get. It may also be insufficient because as we said earlier, our method looks for logical coupling between entities, whereas the given information documents relations which are present in the source code. Therefore, some itemsets may incorrectly be marked as non-relevant, reducing the overall precision. Another thing to notice is that we could not take advantage of the fact that file-level mining can also identify non-source-code files as being part of a concern, as all items in the evaluation sets refer to source code entities.

Second, the selection criteria we used may not be right: we used the F1 score to determine whether an itemset is relevant, but this may not be the right measure. We could have used a parametrised version of this score instead (F_β), which weighs recall more than precision or vice versa. However, weighing recall more than precision will simply give a bias to larger itemsets, and weighing precision more will give a bias to smaller itemsets. It is doubtful whether this really makes for a more accurate selection criterion.

Third, we used only two test subjects to evaluate our results. It would be nice to use other test cases for this as well, but then we should also have cross-cutting concern information on those subjects. Currently, no such information is available (except for some small case studies that did not have enough concerns to be relevant test subjects).

5. CONCLUSIONS AND FUTURE WORK

The central research question of this thesis was "Can we apply frequent itemset mining on version control system data to find cross-cutting concerns in a software system?". In order to answer this question, we developed a tool-chain that can mine a version control system on two different levels of granularity: file-level and method-level. This gave us a list of so-called changesets: entities (files or methods) that were committed simultaneously. By running a frequent itemset mining algorithm on these, we got itemsets consisting of the names of entities (files or methods) that were frequently committed simultaneously. In order to see whether these itemsets can be used to identify cross-cutting concerns, we ran our tool-chain on two different systems (JHotDraw and Tomcat) and compared the results with known sets of cross-cutting concerns. We gave a score to each itemset to indicate how well it represented a concern; we used a score that is commonly used in data mining: the 'F1' metric.

As we cannot be completely sure that the sets of cross-cutting concerns we used for evaluating our approach are correct (in the sense that they describe all concerns in the systems and that they do not contain false positives), we cannot answer the research question with a simple 'yes' or 'no'. Nevertheless, we made the following observations:

- The itemsets we found exhibited low average F1 scores, but the results were always about 3 times better than the scores for randomly generated itemsets.
- Reinforcing changesets improved the F1 scores for file-level mining but not for method-level mining.
- Tweaking the other parameters did not improve the final score most of the time.
- There is no direct relation between the input parameters and the resulting final score. This means that ranking the itemsets is not possible.

The above observations show that while our frequent itemset mining approach is promising, we are also aware of the need to improve our results (and that we should be able to rank the itemsets in order to make the results easier to interpret). In order to do so, we believe that our approach should both be refined and combined with other approaches (see future work below).

Contributions. We made the following contributions:

- We have developed a tool[13] to mine frequent itemsets from VCS data on both file and method level.
- We have performed two case studies and did a thorough evaluation to assess the merits of our approach.

Future work. While we have tested our approach with two systems for which several cross-cutting concerns are known, mining the repositories of more software systems would allow us to be more conclusive about the results.

As we could not predict well what concerns would be identified by our technique, we did not focus on specific concern types. However, making the analysis more specific may improve the results we get. One way to do this would be to use the extra information that method-level mining gives us with respect to the types of changes that were done. For example, we can detect when the `throws` clause of a method has been changed and use this information to focus on the Exception Propagation concern sort [20].

Some of the itemsets we found may have been relevant as cross-cutting concern candidates whereas they were not identified as such. A manual analysis of the itemsets would allow to find out whether our technique has found cross-cutting concerns that have previously not been identified.

Some itemsets contained more items than the itemset it matched best. Although this meant a decrease in precision in our evaluation, it might be that these items were actually relevant. If so, it will be worthwhile to combine our technique with other aspect mining techniques: incomplete concern candidates from another technique can then be expanded automatically by applying our technique.

Acknowledgements.

Part of this research was sponsored by the Center for Dependable ICT (CeDICT), an initiative of NIRICT, the Netherlands Institute for Research on ICT.

6. REFERENCES

[1] Bram Adams, Zhen Ming Jiang, and Ahmed E. Hassan. Identifying crosscutting concerns using historical code changes. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 305–314. ACM Press, 2010.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International*

[13] The tool is available at `http://swerl.tudelft.nl/bin/view/Main/CccFci`

Conference on Management of data, pages 207–216. ACM, 1993.

[3] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proceedings of the International conference on Automated Software Engineering (ASE)*, pages 310–315. IEEE, 2004.

[4] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the International conference on Automated Software Engineering (ASE)*, pages 221–230. IEEE, 2006.

[5] Silvia Breu, Thomas Zimmermann, and Christian Lindig. HAM: cross-cutting concerns in Eclipse. In *Proceedings of the OOPSLA workshop on eclipse technology eXchange*, pages 21–24. ACM, 2006.

[6] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[7] Doug Burdick, Manuel Calimlim, Jason Flannick, and Tomi Yiu. MAFIA: A maximal frequent itemset algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 17(11):1490–1504, 2005.

[8] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. On the use of line co-change for identifying crosscutting concern code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 213–222. IEEE, 2006.

[9] Mariano Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwé. A qualitative comparison of three aspect mining techniques. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 13–22. IEEE, 2005.

[10] Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 189–198. IEEE, 2006.

[11] Margaret H. Dunham. *Data mining: Introductory and advanced topics*. Prentice-Hall, 2003.

[12] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[13] Bart Goethals. *Data Mining and Knowledge Discovery Handbook: A Complete Guide to Researchers and Practitioners*, chapter 17, page 390. Springer Science & Business, 2005.

[14] Lile Hattori and Michele Lanza. On the nature of commits. In *Automated Software Engineering - Workshops (ASE Workshops)*, pages 63–71. IEEE, 2008.

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

[16] Andy Kellens, Kim Mens, and Paolo Tonella. A survey of automated code-level aspect mining techniques. In *Transactions on Aspect-Oriented Software Development IV*, volume 4640 of *LNCS*, pages 145–164. Springer, 2007.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. of the European Conf. on Object-Oriented Programming (ECOOP)*, volume 1241/1997 of *LNCS*, pages 220–242. Springer, 1997.

[18] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[19] Marius Marin, Leon Moonen, and Arie van Deursen. A classification of crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 673–676. IEEE, 2005.

[20] Marius Marin, Leon Moonen, and Arie van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 29–38. IEEE, 2006.

[21] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.

[22] Kim Mens, Andy Kellens, and Jens Krinke. Pitfalls in aspect mining. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 113–122. IEEE, 2008.

[23] Kim Mens and Tom Tourwé. Delving source code with formal concept analysis. *Computer Languages, Systems & Structures*, 31(3-4):183–197, 2005.

[24] Kim Mens and Tom Tourwé. Evolution issues in aspect-oriented programming. In Tom Mens and Serge Demeyer, editors, *Software evolution*, chapter 9, pages 203–232. Springer-Verlag, 2008.

[25] Frank Mulder. Identifying cross-cutting concerns using software repository mining. Master's thesis, Software Engineering Research Group, Delft University of Technology, 2009.

[26] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pages 112–121. IEEE, 2004.

[27] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. LCM: An efficient algorithm for enumerating frequent closed item sets. In *Proceedings of the International Conf. on Data Mining (ICDM)*, 2003.

[28] Charles Zhang and Hans-Arno Jacobsen. PRISM is research in aspect mining. In *Companion to the conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA Companion)*, pages 20–21. ACM, 2004.

[29] Danfeng Zhang, Yao Guo, and Xiangqun Chen. Automated aspect recommendation through clustering-based fan-in analysis. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 278–287. IEEE, 2008.

[30] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

TUD-SERG-2010-032
ISSN 1872-5392