# Identifying Cross-Cutting Concerns Using Software Repository Mining

*Master's Thesis*

Frank Mulder

# Identifying Cross-Cutting Concerns Using Software Repository Mining

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Frank Mulder
born in Rotterdam, the Netherlands


Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

# Identifying Cross-Cutting Concerns Using Software Repository Mining

Author: Frank Mulder
Student id: 1233475
Email: fmuldr@gmail.com

**Abstract**

Cross-cutting concerns are pieces of functionality that have not been captured into a separate module. They form a problem as they hinder program comprehension and maintainability. Solving this problem requires first identifying these cross-cutting concerns in pieces of software. Several methods for doing this 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. We evaluate the results we get from our technique by comparing them with those known concerns. Based on the results of the evaluation, we make some suggestions for future directions in the area of identifying cross-cutting concerns using software repository mining.

Thesis Committee:

| | |
|---|---|
| Chair: | Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft |
| University Supervisor: | Dr. A. Zaidman, Faculty EEMCS, TU Delft |
| Committee Member: | Dr. P. Cimiano, Faculty EEMCS, TU Delft |

# Preface

This thesis presents the work I have done for my graduation project at the Software Engineering Research Group of the Delft University of Technology, which I performed during the academic year of 2008–2009 under the supervision of dr. Andy Zaidman. With this thesis, I will finish my master's degree programme in Computer Science at the EEMCS faculty of the TU Delft.

First of all, I would like to thank my supervisor, Andy Zaidman, for making me enthusiastic about software repository mining and thus greatly helping me in finding a nice topic for my research project. Also thanks for motivating me during the times I got stuck with my research and for encouraging me with kind words every time I handed in pieces of my thesis.

The other two members of my committee also deserve my thanks. I always enjoyed the lectures of Arie van Deursen but he also greatly helped me during my research project. Philipp Cimiano gave me some useful tips during the final part of my thesis project and steered me in the right direction with regard to my thesis writing.

During my studies I have met many people and have made many friends. I had a great time with Ilyaz and Cynthia but also with many other people; listing all their names would make my thesis far too thick.

Especially during my last year I have met many people in the student room where I performed my research. I had a particularly nice time when Leo was there to carry out part of his thesis project and allowed me to practice my Italian with him. He and the other foreign students who spent some time in the student room created an international environment in which I felt like a fish in water. They all taught me at least a few words of their languages, so my thanks go to Arman, Saman and Mani who taught me Persian, Adam who taught me some Polish, and George who taught me some Romanian. I should not forget to mention some Dutch people whom I met there, however: Joost for the nice discussions we had and the good ideas he had with respect to my thesis project and Martin who also provided me with some valuable tips.

Finally I would like to thank my family, in particular my parents, for always supporting me and helping me in becoming the person I am now.

Frank Mulder
Delft, the Netherlands
August 3, 2009

# Contents

# List of Figures

# Chapter 1

# Introduction

In software development, programmers try to achieve a separation of concerns: the implementation of each piece of functionality should reside 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. They cut through the whole system and are called *cross-cutting concerns* [8]. This results in two problems: first, they hinder program comprehension because programmers have to keep track of various concerns while inspecting a piece of code. Second, they decrease maintainability of software since modifying one piece of functionality requires changing code in many places. To be able to solve this problem, we should first *find* those cross-cutting concerns.

## 1.1 Problem Statement

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 techniques to reveal cross-cutting concerns. 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 with the help of data mining techniques. Examples of software repositories include version control systems such as CVS and Subversion and defect tracking systems such as Bugzilla. The research presented in this thesis focuses on mining version control systems (VCS).

Version control systems are commonly used in software development, to facilitate working in teams. They store the files related to a software system in a central place and allow developers to commit changes to them. Although the purpose of version control systems is 'merely' to simplify working in teams on a project, we can exploit hidden relationships in those repositories to support the maintenance of software systems.

In particular, the fact that certain files are often changed simultaneously 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.

If developers commit files that are related to a particular concern in a self-contained transaction, we can use this fact to detect cross-cutting concerns. Doing so is generally accepted as good practice; for example, 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"[1]. Many projects, however, do not have such an official commit policy and even if they have one developers are not forced to adhere to it. Therefore, deviations from this rule should be expected and taken into account when mining cross-cutting concerns from a version control system.

One advantage of using a version control system to mine cross-cutting concerns compared to other techniques is that we can find relations that may not be present in the source code. Traditional techniques rely on things like consistently applied naming conventions or structural relations in source code (such as methods calling other methods). Using the history of software (as recorded in the VCS), on the other hand, can 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" [15]. Exploiting this logical coupling may provide a new perspective on finding cross-cutting concerns in a software system.

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'. This leads us to the central research question of this thesis:

> *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 and should be checked to see if they actually represent cross-cutting concerns. During the literature study preceding this thesis project, we found that many techniques were evaluated in an insufficient way. Mens et al. [29] concluded the same by saying: "Most of the approaches we studied do not provide an empirical validation of their results but rather provide a more incidental validation of their work." Many evaluations are of a subjective nature and there is a persistent lack of quantitative information, which makes comparing techniques difficult. Therefore, particular attention is paid to the evaluation of the 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, but we can also mine on a finer-grained level and investigate which lines in a file have been changed. By mapping source code changes to entities, we can also mine changes to entities such as methods (as in the case of source code written in an object-oriented language).

The research described in this thesis focuses on file-level mining (which only considers file names) and method-level mining (which accounts for additions and modifications of

[1] `http://techbase.kde.org/Policies/SVN_Commit_Policy#Commit_complete_changesets`

methods). It will describe how well both techniques perform in terms of speed, memory usage and result accuracy.

## 1.2 Paper Structure

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

# Chapter 2

# Background and Related Work

The topic of this thesis can be divided into two main fields of research: finding cross-cutting concerns (often called 'aspect mining') and software repository mining. This chapter gives some background information on these subjects, and discusses the research that has already been done in that area.

## 2.1 Aspect Mining

### 2.1.1 Definition

The term *aspect mining* stems from *aspect-oriented programming* [22], a technique that enables us to isolate concerns from the rest of the system in the form of *aspects*: separate modules that implement the functionality of a certain concern.

We prefer to define aspect mining as "the search for cross-cutting concerns in software". Nevertheless, many papers use a slight variation on this definition. For example, some authors have started to distinguish between aspects and 'cross-cuttingness' [29]. Apparently, they use the term 'aspect' to signify the special construct with which concerns are implemented, whereas it appears that originally cross-cutting concerns meant the same as aspects [31].

Some people think that aspect-oriented programming is needed to solve the problem of cross-cutting concerns. They implicitly define cross-cutting concerns as "those things in a software system that should be implemented using aspect-oriented programming (AOP)". Then using the term 'aspect mining' may contribute to the confusion, as it suggests that cross-cutting concerns should indeed be refactored into aspects. We believe that finding cross-cutting concerns is useful, but that not all of them need refactoring into aspects. Some concerns may be eliminated by employing object-oriented refactoring, and others might not be a problem as long as we document them well. To prevent confusion about the meaning of the term 'aspect', we speak of "identifying cross-cutting concerns" rather than "aspect mining" in the title of this thesis.

Other variables in the definition are:

**Degree of automation** Most techniques mine cross-cutting concerns semi-automatically:

a user "has to pre-process the tool's input and/or post-process and analyse its output" (i.e. there is some manual work involved) [29]. However, completely automated or even completely manual methods aimed at finding cross-cutting concerns may also be considered to be forms of aspect mining.

**Stage of development** Instead of looking for cross-cutting concerns in source code (i.e. in the implementation phase), we can also do that in deliverables from earlier stages of development, such as in the requirements or in the architecture description [4].

**Legacy systems** Some papers state that aspect mining deals with "legacy systems" [6, 14] ("large systems that we do not know how to deal with but that are vital to [an] organization" [5]). However, aspect mining may also deal with systems that are actually well designed but for which we still want to identify cross-cutting concerns. Aspect mining can even support software evolution when applied correctly, by noticing cross-cutting concerns just as they are introduced in a system.

**Object-orientation** Most aspect mining solutions target object-oriented systems, but some non-object-oriented systems have been researched as well [1, 9].

In this thesis, we talk about finding cross-cutting concerns semi-automatically in source code written in an object-oriented language (in particular Java).

### 2.1.2 Techniques

As indicated in the introduction, there are several techniques for finding cross-cutting concerns; many of them operate on static data, others operate on dynamic data, and some work with software repository data. Some examples of static techniques follow:

**Identifier analysis** There are a few techniques that analyse the identifiers present in source code. They are "based on the assumption that cross-cutting concerns are often implemented by the rigorous use of naming and coding conventions" [21]. It groups identifiers together that have similar names, or that have a similar meaning. The grouping can be done using, for example, formal concept analysis [37], natural language processing [35] or hierarchical clustering [34].

**Fan-in analysis** This technique is based on the notion that cross-cutting concerns are often implemented using one method that is called from many different places [28]. The number of distinct method bodies that can invoke a certain method is called the fan-in. For each method, the fan-in metric is computed, after which utility methods and methods with a low fan-in are discarded. The remaining set of methods is then manually analysed to find cross-cutting concern candidates.

**Clone detection** Cross-cutting concerns often manifest themselves as code clones: when it is not possible to restrict a concern to one module, developers are "forced to write the same code over and over again" [9]. Therefore, code clone detection can be helpful in finding cross-cutting concerns, and various techniques have successfully put this idea into practice [9, 33].


**Figure 2.1:** The generic software repository mining process

An example of a dynamic technique is using program traces to find recurring execution patterns [6]. For a more detailed description of aspect mining techniques, see the survey by Kellens et al. [21].

Before discussing how software repositories can be used to find cross-cutting concerns, the following section discusses how those repositories can be mined, and what we can achieve by doing so.

## 2.2 Software Repository Mining

Software repositories contain a wealth of hidden information. As said in the introduction, extracting this information is called 'software repository mining'. A selection of software repository mining applications follows:

**Mining co-change** This concerns the question "which entities are commonly changed together?" It can be used to detect logical coupling between modules, which in turn helps to identify architectural weaknesses [17]. It can also be used to predict which entity will be changed after changing another entity, thereby guiding programmers in software development [45].

**Co-evolution of testing and production code** This helps to assess the testing process followed thus far and to monitor if the current testing process matches the intended process [24, 42].

**Project management** Using a software repository, we can for example study the roles of developers, their contributions and their expertise [3, 23, 41].

**Bug prediction** This involves predicting bugs and the time needed to fix them [30, 32].

Figure 2.1 shows how software repository mining works in general. It starts with acquiring data from a *repository*, which can be a version control system but also another type of repository such as a defect tracking system or a discussion archive. In *data acquisition*, the raw data from the repository are transformed into a format that can be processed easily. The *processing* module then typically performs some mining algorithm on the data it gets, after which the results are *presented* to the *user*.

If the repository being mined is a version control system, it can be modelled as a set of transactions, where each transaction consists of entities that have been added, modified or

deleted in the same commit. As said in the introduction, these entities can be files but also finer-grained elements such as lines or words, or syntactic components such as methods.

Data acquisition involves pre-processing the repository data for the next step. For example, CVS does not record which files were committed together, so if we need this information, we will have to reconstruct it. Data acquisition can also comprise "mapping changes to entities", which is needed if we want to analyse entities which are finer-grained than files. Typically, a tool like `diff` is used to find out which parts of a file have changed, after which these are mapped to syntactic components such as methods [44].

The processing step fetches the implicit information contained in the data. To this end, several mining algorithms can be used, such as formal concept analysis and association rule mining. In plain words, *formal concept analysis* is a technique for clustering objects having the same attributes, and clustering attributes corresponding to the same objects (for a more complete explanation, see the paper by Ganter et al. [18]). As for *association rule mining*, most people are familiar with this technique in the context of online stores, where recommendations such as "customers who bought product $x$ also bought product $y$" are given [16]. It consists of two parts: frequent itemset mining and rule extraction. The first part finds sets of items that frequently occur together (such as $\{Bread, Peanutbutter\}$) and the second part creates association rules out of those (such as $Bread \Rightarrow Peanutbutter$). We will see in Chapter 3 that in our tool we will only use the first part, frequent itemset mining.

Finally, there are several ways to present the processed data to the user. For some purposes statistical summaries may be enough, and others may require more sophisticated visualisations such as graphs. Another option is to use an IDE plug-in that lists cross-cutting concern candidates and can highlight related code; this is useful when the mining results should be tightly integrated with the development process. This presentation technique was used in history-based aspect mining, which is discussed in the next section.

## 2.3 History-based Aspect Mining

Not much research has yet been done on using software repository mining to find cross-cutting concerns. Actually, only one research group has done a research 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. Another relevant study has been done by Canfora, Cerulo and Di Penta, in which the evolution of cross-cutting concerns in the application JHotDraw was investigated.

### 2.3.1 Breu et al.

HAM operates on CVS repositories, and uses a tool called APFEL to restore transactions and to find out at which locations method calls have been inserted. Transactions are then defined as sets of pairs $(l, m)$, which represent the insertion of a call to method $m$ in the body of the method $l$.

From this, cross-cutting concern candidates are identified by finding so-called maximal blocks. When considering the insertion of calls to one single method, a maximal block simply consists of all the locations at which those calls were added. However, a cross-

cutting concern may also consist of calls to multiple methods. For example, a call to `lock` will often be followed by a call to `unlock`. This requires a more general definition of a maximal block. Given that a transaction $T$ is a relation between locations $\mathcal{L}$ and methods $\mathcal{M}$ (i.e. $T \subseteq \mathcal{L} \times \mathcal{M}$), a maximal block is defined as a pair $(L,M)$ where the following holds:

$$\begin{aligned} L &= \{l \in \mathcal{L} | (l,m) \in T \text{ for all } m \in M\} \\ M &= \{m \in \mathcal{M} | (l,m) \in T \text{ for all } l \in L\} \end{aligned}$$

Breu et al. only consider blocks with $|L| \geq 7$ (i.e. in which at least 7 method calls were added). These blocks are efficiently found using formal concept analysis (as briefly explained in Section 2.2). Next, they are ranked using an attribute called 'compactness', which is the ratio between the number of locations where calls to one or more methods occurred and the total number of locations where calls to those methods occurred in the history.

The problem with this ranking is that when the introduction of a cross-cutting concern is spread over several transactions, it will be recognised as multiple smaller cross-cutting concern candidates, which will be ranked low. Therefore, Breu et al. exploit locality to reinforce aspect candidates (i.e. to merge them into one aspect candidate, which will be ranked higher). They recognise two types of locality: temporal locality (aspect candidates may appear in transactions that are close in time) and possessional locality (aspect candidates may be created by one developer but committed in different transactions). By merging the locations at which the calls for a certain method are inserted, they obtain a reinforced aspect candidate.

Finally, they combine simple candidates to form complex candidates if two candidates cross-cut exactly the same locations. The resulting cross-cutting concern candidates are presented by means of an Eclipse plug-in, in which a list of candidates is given that is linked to the source code; that way, the various candidates can be easily explored.

As Breu et al. tried to achieve the same goal as we do (that is, finding cross-cutting concerns using a software repository) one might wonder: what distinguishes their technique from ours? One important thing to notice is that their technique is very similar to what Marin et al. [28] did with fan-in analysis: both consider the number of calls to certain methods (i.e. the fan-in). Fan-in analysis considers the fan-in values of the methods in a certain snapshot of the source code, whereas 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 thing is that their technique falls into the same trap as many other techniques: their evaluation is subjective and makes comparing the performance of their technique to that of other techniques difficult. This is something that we address in our research as well.

### 2.3.2 Canfora et al.

The work of Canfora et al. is particularly interesting because they wrote a paper titled "On the Use of Line Co-change for Identifying Crosscutting Concern Code" [11]. At first, this sounds like they did the same as we are trying to do in this thesis: exploiting co-change information from a software repository to find cross-cutting concerns. The title is a bit misleading, however, as their technique does not identify any cross-cutting concerns at all. Instead, their technique is based on an already known set of cross-cutting concerns in JHotDraw, for which they try to find corresponding transactions from the version control system afterwards. The decision whether a specific set of transactions represents a cross-cutting concern is based on the lines that were changed in those transactions. The union of those lines is compared with the known lines of a cross-cutting concern to see how well they match.

As this technique relies on a known set of concerns, it cannot identify them on its own. However, this research provides some nice insights into how cross-cutting concerns evolve over time (on which they wrote another paper [10]). It appears that cross-cutting concerns are often introduced in one transaction and then extended in later transactions (an observation that was also made by Breu et al. [7]). However, this was based on a specific selection of concerns (most of which are related to design patterns) so it remains unclear whether this holds for all types of concerns.

The known set of cross-cutting concerns they used is actually based on the results of fan-in analysis as performed by Marin et al. [28], which they annotated with line numbers. Luigi Cerulo (who is part of the research group that did the research we just described) kindly provided us with these data, which we used to evaluate our own approach.

# Chapter 3

# Tool-chain Structure and Implementation

This chapter 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. [27] propose a common framework for aspect mining, which "allows for consistent assessment, comparison and combination of aspect mining techniques". It requires one to define 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 to choose from has been made by the same authors [26]. 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 and the results. Our technique can be explained in the terms of that framework with the following definitions:

**Search goal** 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 [26] 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. At this point, however, we do not know exactly whether such changes are frequently made in practice. That is, a concern may be cross-cutting, but as long

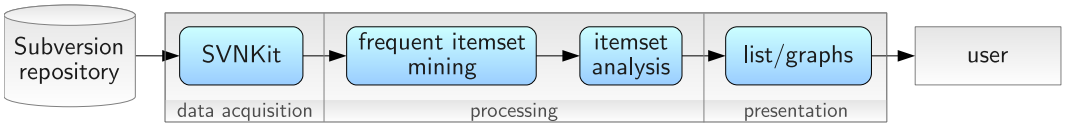


**Figure 3.1:** Tool-chain structure.

as it is not changed, we will not detect it. To be able to find out which concerns are often modified, we do not limit our technique to specific concern sorts.

**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 we analyse, we use the so-called F1 measure to determine how well an itemset represents a cross-cutting concern. This measure will be explained in detail in Section 3.5.2. For each itemset we determine the best matching concern, i.e. the one with the maximal F1. To get an overall score of how well the complete set of itemsets matches the concerns in the analysed subject we take the average of these values, which gives us the average maximal F1.

## 3.2 Tool-chain Structure

Section 2.2 explained how software repository mining tool-chains work in general. Our tool-chain contains the same elements of data acquisition, processing and presentation. Figure 3.1 shows more specifically what steps are taken in the process. It acquires data from a Subversion repository using the SVNKit library[1], and outputs 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, although we will also use graphs to visualise the results.

The tool-chain has been implemented in Java, with each module implemented as a separate class (which gets input data and a configuration, and transforms it to produce data which serves as input for the next module), facilitating extension and modification of the tool-chain. However, frequent itemset mining algorithms are usually implemented in C++, so we call them externally (still, we implemented it such that various algorithms can easily be incorporated in the tool-chain).

Our tool-chain is limited to analysing systems written in Java. As indicated in Chapter 1, we perform the mining process on two levels of granularity: on file-level and on method-level. Both are in a certain way tied to a specific source language. When using Java,

[1] `http://svnkit.com`

file-level mining practically means class-level mining as that language forces one to write each class in a separate file (except for inner and anonymous classes). Strictly speaking, file-level mining is not bound to be used with Java but when mining source code written in a language that does not impose these conventions we cannot speak about class-level mining any more. On the other hand, method-level mining is tied more strictly to a given source language, as the parser analysing the source files is made for a specific language. In particular, when we speak of "method-level mining", we restrict ourselves to source code written in an object-oriented language. However, we could extend our method-level miner by allowing different parsers to be used, thereby allowing source code written in different object-oriented languages to be analysed. With that said, we limit ourselves to analysing Java code for the scope of this research, but we expect that a modified version of our tool-chain could be used to analyse other systems as well.

The two levels of granularity we just mentioned each have their own pros and cons:

**File-level mining** This only deals with the names of files that were frequently changed together. Advantages of this are that it requires little effort to extract the data we want from the repository, and the number of entities to analyse will be relatively small (compared to method-level mining), with short execution times as a result. Another advantage is that non-source code files (such as 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, possibly making the results of the analysis less useful than more precise methods (largely due 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 in order to be able 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 more precise than those found with file-level mining, probably 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 experiments. 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 a Subversion repository. Subversion was designed as the successor to CVS, another popular version control system. One important advantage of Subversion over CVS for data miners is that it stores files

that were committed simultaneously in one transaction, whereas CVS creates one separate transaction for each file that is committed. This is important because we rely on the knowledge that files belong to the same transaction. Also, CVS repositories can be converted to a Subversion repository by using `cvs2svn`[2], a tool that restores transaction information and "deals with many CVS quirks". This practically means that we can analyse both CVS and Subversion repositories; in fact, the repositories of many projects have already been converted to Subversion using `cvs2svn`, so this decision is expected to result in a wide applicability of our tool-chain.

We use SVNKit, a Java Subversion library, to fetch log entries from a Subversion repository. A log entry consists of the revision number, time-stamp, author and log message for each transaction, along with the changeset: the files that were added, modified or deleted. Similar to what Breu et al. [7] 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. Formally, the entities of transactions $T_1$ and $T_2$ are combined if

$$|time(T_2) - time(T_1)| \leq interval$$
$$\text{and} \qquad author(T_1) = author(T_2).$$

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 uninteresting 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.

As the frequent itemset mining module expects to have only integer values as input, the entity names are mapped to unique identifiers. These are stored in a bi-directional hash map, so the identifiers can be translated back to entity names later on.

In addition, we keep track of files that have been renamed, moved or copied using Subversion's `copy` command. We do this by reusing the identifier of the copied path for the new path.

Some more work has to be done when performing method-level mining. In order to find out which methods have been added or modified in a certain transaction, we have to find out which parts of a file have been changed. This is done using a modified version of DiffJ[3], which is like the Unix program `diff`, but specifically for Java code. We had to modify the source code of this program because it only reports changes in the highest node in an abstract syntax tree; for example, if a new class was added, it would not report the methods inside it as being added. Some additional modifications were needed to make sure it would output the changes in a format usable by our tool-chain, outputting the full name of each method, including the package name and the names of the enclosing class and any inner classes. In this case, the change sets contain the full names of each method (as opposed

[2] `http://cvs2svn.tigris.org`

[3] `http://www.incava.org/projects/java/diffj/`

to file-level mining, where they consist of file names). As soon as we have collected these change sets, we can analyse them. This is explained in the next section.

## 3.4 Frequent Itemset Mining

A common way to mine patterns in a database is *frequent itemset mining* (FIM)[4]. 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, to generate association rules in the second step. 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 it is not needed to generate rules from that, although we do assume that the frequent occurrence of sets of entities implies that there is a correlation between them. Note that this module can be used without modification for both file-level and method-level mining, although the latter will probably have to handle a larger set of input data, increasing time and space requirements. 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 call $X \subseteq I$ an itemset. Further, define database $D$ as a set of transactions: $D = \{t_1, t_2, \ldots, t_n\}$, 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, 16].[5]

Then $X$ is called a *frequent itemset*[6] 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 $I$ (all possible itemsets that could be generated using $I$), i.e. $\mathbf{FI} \subseteq 2^I$.

[4] This seems the right moment to acknowledge the fact that 'itemset' is not an actual English word. It obviously just means a "set of items" and could be written as 'item set' but we have chosen to maintain consistency with FIM literature and to write it as one word (for a similar reason we also used 'changeset' in the previous section).

[5] Sometimes, the support of $X$ is (implicitly) defined as the number of transactions that contain $X$ (i.e. $|t(X)|$), and not as the fraction. In this paper we will stick to using the fraction, and use "number of occurrences" when we mean $|t(X)|$.

[6] Sometimes, the term *large itemset* is used, but we will avoid to use this term, as the word 'large' could be mistaken to refer to the cardinality of the set, whereas it refers to the number of occurrences.

**Table 3.1:** Example database with 4 transactions.

| Transaction | Items |
|---|---|
| $t_1$ | $\{a\}$ |
| $t_2$ | $\{a,b,c\}$ |
| $t_3$ | $\{b,c,d\}$ |
| $t_4$ | $\{b,c,d\}$ |

**Table 3.2:** Support of all itemsets that can be generated using the items in the database of Table 3.1.

| Itemset | Support | Itemset | Support |
|---|---|---|---|
| $\{\}$ | 1 | $\{b,c\}$ | 0.75 |
| $\{a\}$ | 0.5 | $\{b,d\}$ | 0.5 |
| $\{b\}$ | 0.75 | $\{c,d\}$ | 0.5 |
| $\{c\}$ | 0.75 | $\{a,b,c\}$ | 0.25 |
| $\{d\}$ | 0.5 | $\{a,b,d\}$ | 0 |
| $\{a,b\}$ | 0.25 | $\{a,c,d\}$ | 0 |
| $\{a,c\}$ | 0.25 | $\{b,c,d\}$ | 0.5 |
| $\{a,d\}$ | 0 | $\{a,b,c,d\}$ | 0 |

### 3.4.2 Example

To see how frequent itemset mining works, consider the example database in Table 3.1. The database contains 4 transactions, which are subsets of $I = \{a,b,c,d\}$.

Table 3.2 lists the items in the power set of $I$ together with their support. First, note that the support of the empty set is 1; this is obviously because the empty set always is a subset of any other set (therefore, this information is not very interesting, and sometimes omitted by FIM algorithms). Four other sets have a support of 0; this is because the items in those sets do not occur simultaneously in any transaction. These will normally not appear in the output of an FIM algorithm, as they are not frequent at all. Of the other sets, only those with a support equal to or greater than the given minimum support will appear in the result. It depends on the purpose we have with mining itemsets which of these are interesting. This again poses requirements on the algorithm we choose to perform the itemset mining with; those are discussed in the next section.

### 3.4.3 Algorithm requirements

In association rule mining, the most interesting itemsets are those with a large support and confidence. Support was defined in Section 3.4.1 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 interesting 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 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) [20]. 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 $2^m$ [16], where $m$ is the size of the largest transaction in the database, so the occasional dense data tremendously hampers the performance of this algorithm.

What does this mean in practice? We tested an implementation[7] of the algorithm using 436 changesets (on file-level) from the Subversion repository of JHotDraw. With the minimum number of occurrences set to 8, the algorithm already took almost 9 minutes to execute, while using more than 1500 MB of RAM at a certain moment.[8] With a minimum number of occurrences of 7, the program filled the entire 16 gigabytes of RAM that were available and was killed by the operating system, so it could not complete. As we also want to investigate itemsets with low support values, using Apriori is not an option for us.

There are lots of other frequent itemset mining algorithms, but before we consider them, 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. For example, take a look at Table 3.2: when we know that there is an itemset $\{b,c\}$ with support 0.75, then the information that the itemsets $\{b\}$ and $\{c\}$ exist (with the same support) is not interesting and could be left out. It appears that generating only these interesting 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.

### 3.4.4 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) [19]. Formally, an itemset $X$ is

[7] The implementation of Bart Goethals, to be found at `http://www.adrem.ua.ac.be/~goethals/software/`.

[8] The tests were performed on a 2.33 GHz CPU core.

**Table 3.3:** Result of frequent closed itemset mining on the data of Table 3.1 for a minimum support of 0.5.

| Itemset | Support |
|---|---|
| $\{\}$ | 1 |
| $\{a\}$ | 0.5 |
| $\{b,c\}$ | 0.75 |
| $\{b,c,d\}$ | 0.5 |

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) [38]. Call the set of frequent closed itemsets **FCI**, then it holds that **FCI** $\subseteq$ **FI** (in practice, **FCI** is orders of magnitude smaller than **FI**). When performing frequent itemset mining on our example database (Table 3.1), it will produce the output shown in Table 3.3. Compared to the result of frequent itemset mining on the same data with the same minimum support, the items $\{b\}$ and $\{c\}$ are left out because they have the same support as their superset $\{b,c\}$; the same holds for $\{d\}$, $\{b,d\}$ and $\{c,d\}$, because their superset $\{b,c,d\}$ has the same support. However, $\{a\}$ is not left out because it has no frequent superset, and $\{b,c\}$ remains because it has a higher support than its superset $\{b,c,d\}$.

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. The results are shown in Table 3.4. As can be seen, the fastest implementation completes in only 3 seconds. Compare this to Apriori, which could not complete even though a much higher minimum support had been set. Clearly, it is worthwhile to use frequent closed itemset mining, and in particular the LCM algorithm seems to perform quite well for our input data (we also considered to evaluate other algorithms like CHARM and CLOSET, but there were no publicly available implementations of them).

LCM stands for Linear time Closed itemset Miner. It has been designed in such a way that it only generates frequent closed itemsets, in contrast to other algorithms, which basically enumerate frequent itemsets and then prune away unnecessary sets. This means that the algorithm is linear in the number of frequent closed itemsets; hence the name. Also, several techniques are used to speed up computation, in particular a technique which adjusts to parts of the input being dense or sparse. This is probably very worthwhile since our input data is also very mixed in this respect, as discussed in Section 3.4.3. For a more detailed explanation of the algorithm, the reader is referred to the various papers on LCM [38, 39, 40].

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

**Table 3.4:** Running times of various frequent closed itemset mining implementations

| Implementation | Running time (seconds) |
|---|---|
| MAFIA[a] | 21770 |
| Yasuo Tabei's LCM[b] | 558 |
| FPclose[c] | 11 |
| LCM version 1[d] | 10 |
| LCM version 4[d] | 3 |

[a] Run with the `-fci` option; available from `http://himalaya-tools.sourceforge.net/Mafia/`.

[b] Available from `http://www.cb.k.u-tokyo.ac.jp/asailab/tabei/Lcm/lcm.html`.

[c] Retrieved from `http://fimi.cs.helsinki.fi/src/`.

[d] Two different versions of the implementation by the original authors of the LCM algorithm, both available at `http://research.nii.ac.jp/~uno/codes.htm`.

imposed on the support of those supersets). This means that a maximal frequent itemset mining algorithm would generate all but one of the sets shown in Table 3.3 when run on our example database: the set $\{b,c\}$ would be left out, as it has the frequent superset $\{b,c,d\}$ (even though its support is lower). As we said in Section 3.4.3, we are 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 for our purpose.

## 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 determine criteria to evaluate whether a candidate does indeed represent a cross-cutting concern.

### 3.5.1 Selecting Cross-Cutting Concern Candidates

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

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

**Cardinality** Itemsets containing very few items may be irrelevant, as well as itemsets containing very many items.

**Lift** The lift measure (sometimes called 'interest') 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 interesting an itemset is.

**Changeset size** As said in Section 3.3, we can discard very small and/or very large changesets to limit the input of the mining module, in order 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.

Note that the last two criteria are not selected in the itemset analysis step, but earlier in the tool-chain (in the data acquisition step).

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

### 3.5.2 Evaluation Criteria

For evaluating our technique, we should define a way to determine how well a certain itemset matches a cross-cutting concern. To make sure our results are reproducible and to facilitate comparison with other techniques, we decided to avoid manual assessment and to do an automated evaluation (as stated in Section 1.1). Therefore, 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 that data, we can determine for each itemset how much it covers a cross-cutting concern (i.e. the recall) and how many of the file names in the itemset do actually belong to that concern (i.e. the precision). 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 the recall would be very low. Therefore we use a combination of precision and recall: the F1 score, a measure which is commonly used in information retrieval and which is the harmonic mean of precision and recall [25]. This can formally be defined as follows:

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

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 interesting 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 simple: a CSV (comma-separated value) file is generated, which contains the itemsets along with the transactions they are based on. If needed, this file can be used to generate graphs to get various views on the itemsets.

# Chapter 4

# Experiment Results

This chapter describes the results of running our tool-chain (which was described in Chapter 3) 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. Even those results are usually not publicly available. Marin et al. [28] have put effort into "setting up a web forum[1] where aspect mining researchers can exchange and discuss aspect candidates found in (open source) software systems". Only the results from their own research (mainly from fan-in analysis) are available there, unfortunately, and contributions from others have failed to appear. 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 and to compare the resulting itemsets with all known cross-cutting concerns we could find for these subjects. The next sections will discuss the results for each application in detail.

[1] `http://swerl.tudelft.nl/amr/`

## 4.2 Case Study: JHotDraw

### 4.2.1 Application Overview

JHotDraw is a Java GUI framework for technical and structural graphics[2]. 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 also makes it a somewhat 'artificial' subject.

The repository of JHotDraw is currently at revision 544 and its history started on 8 August 2002. 11 different developers have committed transactions and the application is still being actively developed.

### 4.2.2 Evaluation Set

In Chapter 2 we indicated that Canfora et al. [11] sent us the list of cross-cutting concerns they had for JHotDraw. It appeared that this list was identical to the list given on the aforementioned web site, although it contained not only 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 wondered if other authors had published their results for JHotDraw as well. The results for the dynamic analysis experiment with the Dynamo tool by Tonella and Ceccato [36] were publicly available[3]. The paper by Ceccato et al. [12] also mentions identifier analysis as one of the techniques tested on JHotDraw. Mariano Ceccato kindly provided us with the results of this experiment. The results of Zhang's PRISM Aspect Miner [43] were publicly available[4] 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 as reproduced in Appendix A.

The studies from which we use the results were all done on JHotDraw version 5.4b1. In the meantime, its version number has progressed to 7.0.8. However, the package structure of JHotDraw has changed a lot during that time, so therefore we decided to only mine the part of repository corresponding to the history until JHotDraw version 6, which has the same package structure as version 5.4b1. This part consists of 172 revisions.

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

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

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

### 4.2.3 Results for File-Level Mining

When running our tool on the repository of JHotDraw while only considering file names, we get 828 itemsets, with an average maximal F1 of 0.36 (running time: 3 seconds[5]). Before drawing conclusions about this score, we can try to improve it by tweaking the criteria mentioned in Section 3.5.1: support, cardinality, lift, changeset size and reinforcement interval. For each of these criteria (except for the last one) we can find appropriate constraints: minimum and maximum values that lead up to a good average maximal F1, which means that we have 9 different input parameters for our tool. 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 (i.e. with the interval set to 0). 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, take a look at Figure 4.1. That graph shows for each number of occurrences and size the maximal F1 score for the itemset that best matched

[5] All experiments were performed on a machine with 16 gigabytes of RAM and two quad core Intel Xeon E5345 processors (2.33 GHz) of which at most two cores were actually used by our tool.

a concern (we have chosen to limit the 'cardinality' axis to only 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. If this score is higher than or equal to 0.5, an abbreviation of the concern name is shown next to the dot. The first letter corresponds to the reference set from which we got the data about that concern: I stands for identifier analysis, D for dynamic analysis and F for fan-in analysis. The complete list of concerns with their abbreviations can be found in Appendix A. We notice the following things in this graph:

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. Similarly, itemsets that occur frequently usually have a small cardinality.
3. 'Sweeps' of similar concerns occur in various places. For example, IB17 can be found at (#occurrences, cardinality) coordinates (2,20), (3,18), (3,16), (4,15), etc.
4. 'Interesting' concerns are not really 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 simply because we instructed the mining algorithm to do so: itemsets with less than 2 items are not relevant because we want to see files that have been committed together, not single files that happen to be committed frequently. Itemsets that occur less than 2 times are not relevant because we want to see which files have frequently been changed together, not only once or never at all.

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 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 (this confirms the similar observation made by Hattori and Lanza [20] which was mentioned in Section 3.4.3).

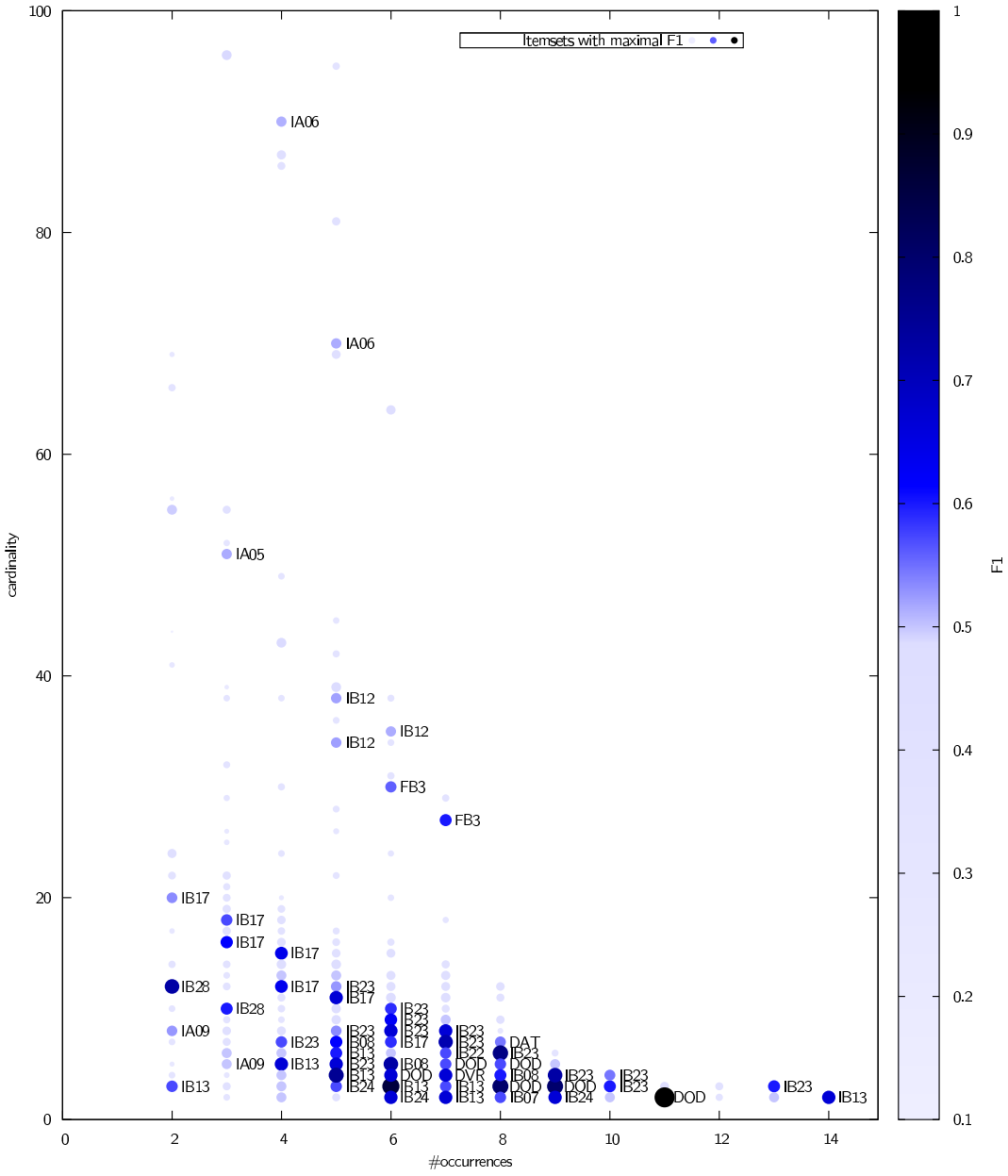


**Figure 4.1:** Itemsets with maximal F1 for the given cardinality and number of occurrences (JHotDraw, file-level). Colour intensity and size represent the F1. Itemsets with F1 $\geq 0.5$ are annotated with the corresponding concern identifier.

**Table 4.1:** Top 7 itemsets for JHotDraw on file-level

| #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] |
| 6 | 3 | 0.86 | IB13 | [11, 12, 39, 46, 57, 76] | [application/DrawApplication.java, contrib/TextAreaFigure.java, contrib/html/HTMLTextAreaFigure.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] |
| 5 | 4 | 0.75 | IB13 | [12, 39, 46, 57, 76] | [application/DrawApplication.java, contrib/TextAreaFigure.java, contrib/html/HTMLTextAreaFigure.java, standard/PeripheralLocator.java] |

To understand the third point, it helps to look at the concerns listed in Table 4.1. For each itemset, it lists the number of occurrences and the cardinality (which corresponds to the number of changesets and the number of itemsets, respectively). It also lists the best-matching concern for each itemset and the corresponding F1 value. The changesets column indicates in which changesets the items in the itemsets occurred.

In that table we can see multiple itemsets that match the DOD (Manage figures outside drawing) concern. The attentive reader will have noticed that all of them contain the two items that the first itemset consists of, and the others add an extra item to it. 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 3.4.4): 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 currently best-matching itemset (because it has frequent supersets).

One might suggest to discard the supersets of the first itemset as 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, in Figure 4.1 we can see that IB28 at (2,12) has a higher F1 than at (3,10). Moreover, the figure shows that a higher support does not necessarily mean that an itemset is more interesting.

This leads us to the fourth point, as we would like to find a relation between the input parameters and the resulting 'interestingness'. However, such a relation does not seem to exist. Although most interesting itemsets seem to have a low cardinality, this holds for the itemsets in general. Consequently, restricting the cardinality to low values does not really help in improving the final 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 final score of 0.42, indeed just a little improvement over the score of 0.4 without these restrictions.

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 that we have not explored yet: 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 interesting). By substituting the support for the (supposedly more accurate) lift, we get the results shown in Figure 4.2.

One striking thing in this figure is that the more 'interesting' itemsets (with a high F1) have a low lift. This is the opposite of the expectation we just formulated. The same figure can help us in understanding this: it suggests a logarithmic relationship between lift and cardinality. This is not very surprising as the support of individual items is usually similar and also very low. Taking the product of these values will lead to an exponential increase of the lift, proportional to the cardinality of the itemset. As we showed that most interesting

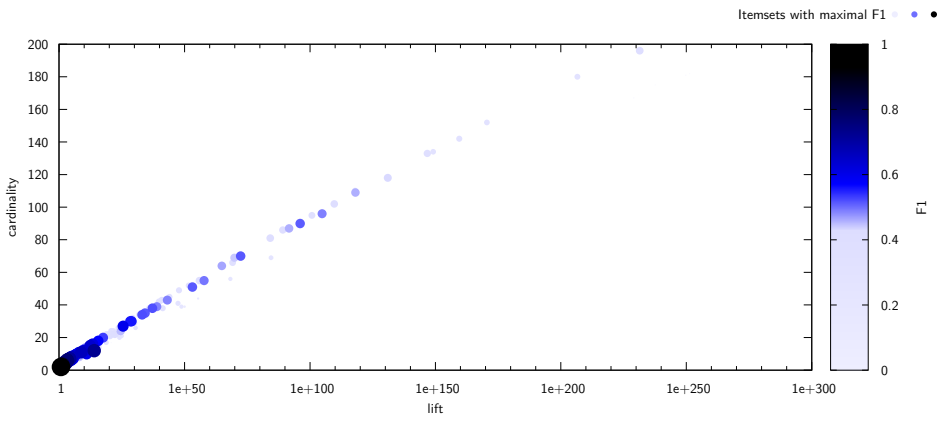


**Figure 4.2:** Itemsets with maximal F1 for the given cardinality and lift (the lift axis is logarithmic).

itemsets have a low cardinality this explains why most interesting itemsets are on the left side of the figure.

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 interesting itemsets. This is something that should still be investigated.

We have tried our best to achieve a high maximal F1 for the given input: with reinforcement enabled, we could reach a score of 0.40. By putting restrictions on other parameters, we could still somewhat 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 interesting 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 given 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 to get a representative final score for the random case. 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. Let us now see how method-level mining performs.

### 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.

For the reinforcement interval we did a similar test as with file-level mining, but this time the results were quite different: the best score we could get with reinforcement was scarcely higher than without (and after rounding the score was again 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 final scores were always lower than without doing so.

Restricting the itemset cardinality could help somewhat, just like in the case of file-level mining, but the best scores were achieved by setting the minimum cardinality to 20, which means again that some relevant itemsets were discarded. This can be seen in Figure 4.3: only the itemsets representing the FM (Command) concern are left, and some itemsets with higher F1 values than all of the itemsets with cardinality $> 20$ are 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. As the itemsets that occurred twice do not seem to be very relevant, we can set the minimum number of occurrences to 3, but this only leads to a slight improvement in the final score: 0.16.

We also did a small test with the lift but it turned out again 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.

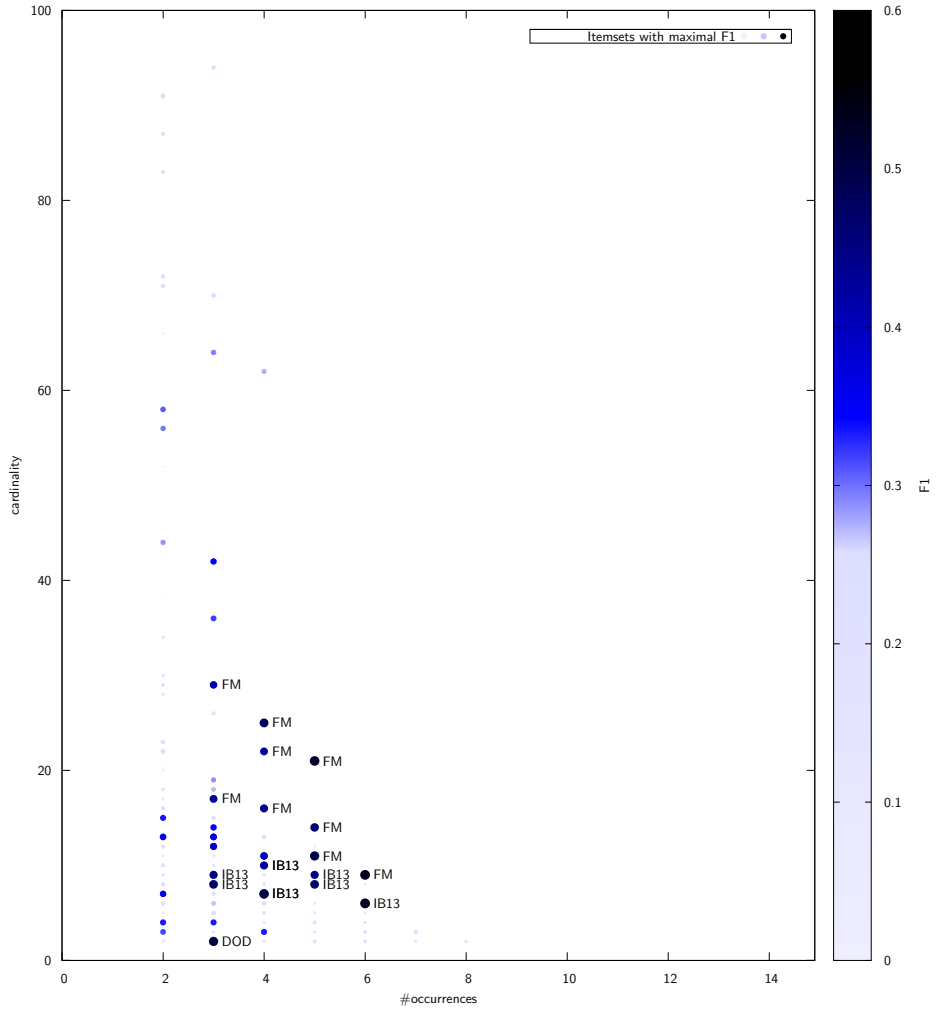


**Figure 4.3:** Itemsets with maximal F1 for the given cardinality and number of occurrences (JHotDraw, method-level). Colour intensity and size represent the F1. Itemsets with F1 $\geq 0.4$ are annotated with the corresponding concern identifier.

## 4.3 Case Study: Tomcat

### 4.3.1 Application Overview

Apache Tomcat (abbreviated to 'Tomcat' in the rest of this document) is "an open source software implementation of the Java Servlet and JavaServer Pages technologies"[6]. It is developed by the Apache Software Foundation, of which the repository is currently at revision 796310. It contains a lot more projects than Tomcat alone and a large part of it consists of imported CVS repositories, some of which date back to 1994.

### 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 their 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. This is demonstrated in Figure 4.4, which shows that many itemsets with a low number of occurrences are still relevant.

The result for the random case was 0.0454, 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

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

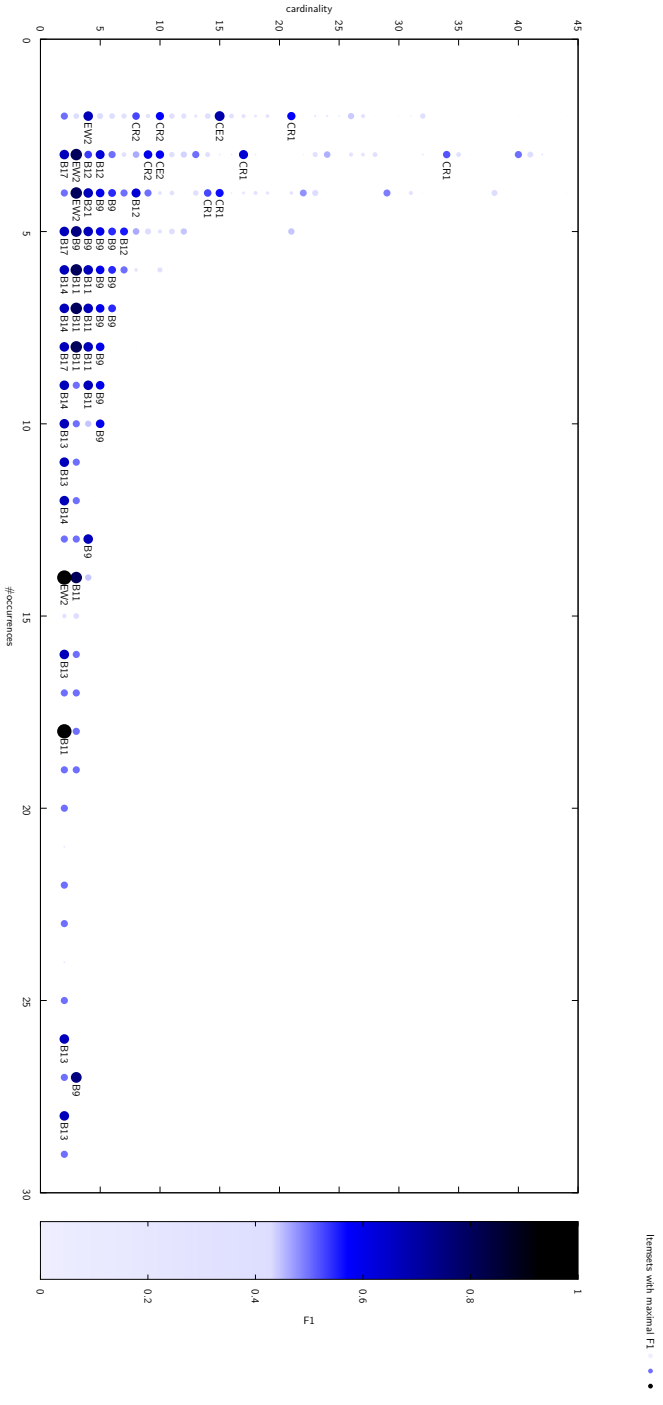


**Figure 4.4:** Itemsets with maximal F1 for the given cardinality and number of occurrences (Tomcat, file-level). Colour intensity and size represent the F1. Itemsets with F1 $\geq 0.5$ are annotated with the corresponding concern identifier.

before would take a very long time. For these intervals we did not get higher F1 scores than without reinforcement. This result is the same as what we got when doing method-level mining on JHotDraw.

This time, however, constraining the changeset size did improve the final score significantly: from 0.02 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.

For the other parameters, take a look at Figure 4.5. It is interesting to see that there are many matches for B9 (an instance of the Consistent Behaviour concern type) in the upper part of the graph and several matches for other concerns in the lower part which are separated by a gap. If we choose to focus the itemsets that matched B9 by setting the minimum cardinality to 50, we can indeed increase the final score up to 0.10. When we combine this with the previously mentioned constraints on the changeset size, which leave almost only itemsets matching B9, 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 toolchain 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.

## 4.4 Threats to Validity

Some factors may threaten the validity of the results presented above. First of all, we are not sure if 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 in the analysed subjects. However, it is based on research that has previously been done on cross-cutting concerns in those systems, so it is probably 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-interesting, 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 we used are source code entities.

Second, the selection criteria we used may not be right: we used the F1 score to determine whether an itemset is interesting, but this may not be the right measure. We could have used a parametrised version of this score instead ($F_\beta$), 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. At the time of the experiments such information was not available (except for some small case studies that did not have enough concerns to be relevant test subjects).

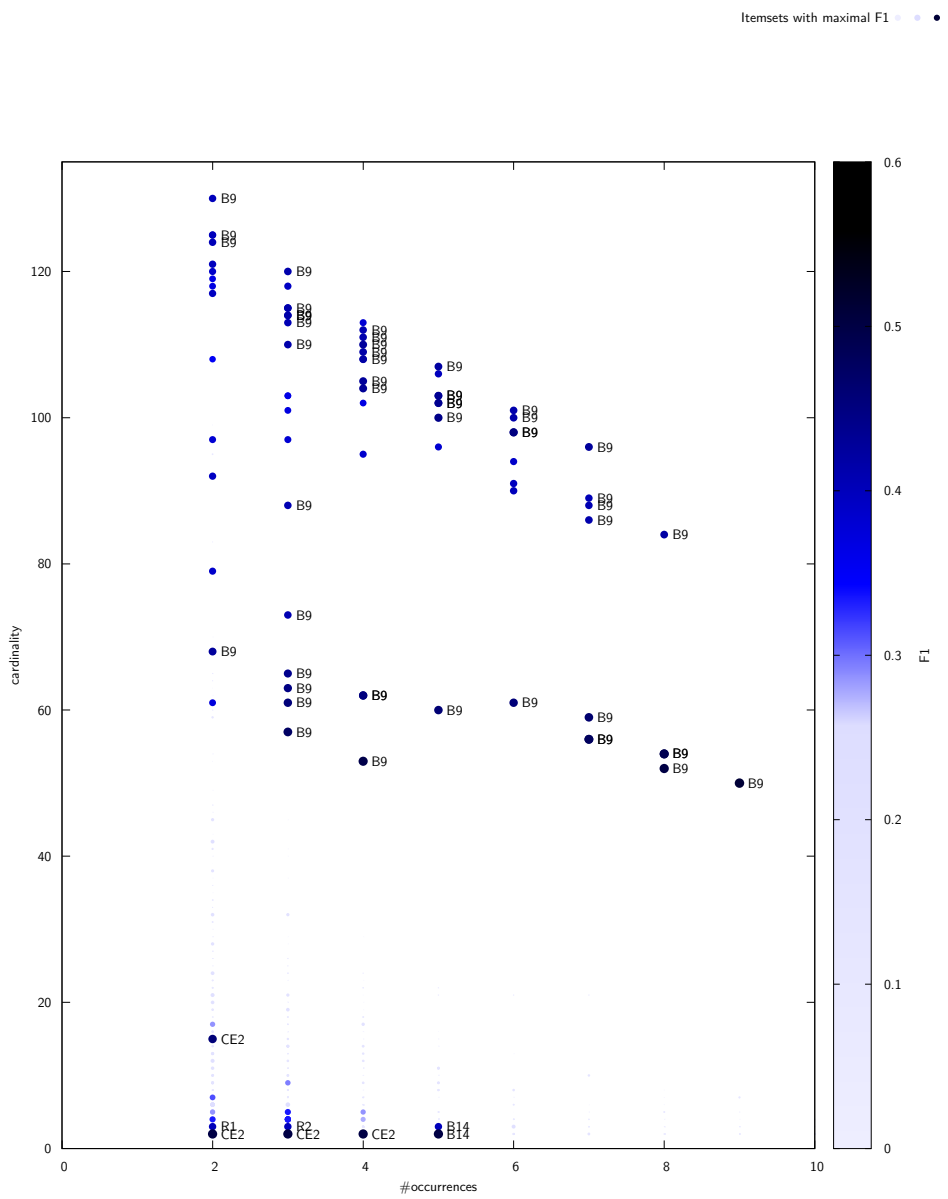


**Figure 4.5:** Itemsets with maximal F1 for the given cardinality and number of occurrences (Tomcat, method-level). Colour intensity and size represent the F1. Itemsets with F1 $\geq 0.4$ are annotated with the corresponding concern identifier.

# Chapter 5

# Conclusions and Future Work

## 5.1 Conclusions

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, meaning that for each transaction in the version control system we considered file names and method signatures, respectively. This gave us a list of so-called changesets: entities (files or methods) that were committed simultaneously. We implemented reinforcement, which means that we could combine changesets if they were committed close in time by the same developer (to compensate for entities that had been committed separately but were still related). 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. In addition, we made it possible to adjust the operation of the tool-chain by tweaking various parameters: reinforcement interval, minimum and maximum changeset size, minimum and maximum itemset cardinality, minimum and maximum support and minimum and maximum lift.

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 during our research:

- 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 (combining) 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.

Still keeping in mind that no firm conclusions can be drawn, the above observations (especially that we cannot rank the itemsets and that they have low scores) lead us to believe that frequent itemset mining alone is probably not very suitable to identify cross-cutting concerns from a version control system. There are still some points to investigate in this area of research; we discuss these after listing the contributions we made with our research.

## 5.2 Contributions

With our research we have made the following contributions:

- We have developed a tool to mine frequent itemsets from version control system data on both file and method level. The tool is very flexible in the sense that it can be adjusted using various parameters. The tool will be publicly available at `http://swerl.tudelft.nl/bin/view/Main/CccFci`.
- We have performed two case studies and did a thorough evaluation to assess the merits of our approach.

## 5.3 Future Work

Based on the research we have done, we can make the following suggestions for future work:

**Evaluate with more systems** We have tested our approach with two systems for which several cross-cutting concerns were known. It would be nice to also test it with other systems, as soon as the concerns in those systems become known and are publicly available. This would allow one to be more conclusive about the results.

**More specific mining** 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.

**Manual analysis** We did an automated evaluation as this would make our tests repeatable and would facilitate comparison with other techniques. However, some of the itemsets we found may have been relevant as cross-cutting concern candidates whereas

they were not identified as such. By doing a manual analysis of the itemsets, we can find out whether our technique has found cross-cutting concerns that have not been identified by other techniques.

**Combining with other techniques** 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 another aspect mining technique: incomplete concern candidates from another technique can then be expanded automatically by applying our technique. A similar thing was done with identifier analysis in the research of Ceccato et al. [13]: identifier analysis alone was not precise enough for the purpose of finding cross-cutting concerns but combining it with other techniques almost always improved the results. Therefore, we think it is a very promising idea to do the same with our technique.

**Devise probability of entities being committed together** When discussing the results of changing the lift parameter, we found that the expected probability of entities being committed together was not accurate enough. Creating a better model than just assuming that entities are independent of one another may help us in devising a better measure to determine how interesting an itemset is.

**Different repository types or source languages** Our technique focused on Subversion repositories and Java source code. As many software projects are now moving towards more modern, distributed version control systems such as Mercurial and Git, it will be interesting to see if those repositories contain extra information that we can use to identify cross-cutting concerns.

# Appendix A

# Concerns in JHotDraw and Tomcat

Here we list the cross-cutting concerns in the subjects that we analysed. In the fan-in analysis results, the numbers between parentheses are instance numbers (which are given because one concern type can have multiple instances).

## A.1 Concerns in JHotDraw

### A.1.1 Concerns Identified by Dynamic Analysis

| ID | Concern name |
|---|---|
| DAT | Add text |
| DAU | Add URL to figure |
| DBF | Bring to front |
| DCE | Command executability |
| DCF | Connect figures |
| DCT | Connect text |
| DFC | Manage figure changed event |
| DFU | Figure update |
| DGA | Get attribute |
| DMF | Move figure |
| DMH | Manage handles |
| DOD | Manage figures outside drawing |
| DP | Persistence |
| DSA | Set attribute |
| DSB | Send to back |
| DU | Undo |
| DVR | Manage view rectangle |
| DV | Visitor |

### A.1.2 Concerns Identified by Fan-in Analysis

| ID | Concern name |
|---|---|
| FA | Adapter(1) |
| FB1 | Consistent Behavior(1) |
| FB2 | Consistent Behavior(2) |
| FB3 | Consistent Behavior(3) |
| FB4 | Consistent Behavior(4) |
| FB5 | Consistent Behavior(5) |
| FB6 | Consistent Behavior(6) |
| FB7 | Consistent Behavior(7) |
| FB8 | Consistent Behavior(8) |
| FB9 | Consistent Behavior(9) |
| FB10 | Consistent Behavior(10) |
| FB11 | Consistent Behavior(11) |
| FC | Composite(1) |
| FD | Decorator(1) |
| FE | Exception Handling(1) |
| FM | Command(1) |
| FO | Observer(1) |
| FP | Persistence(1) |
| FU | Undo(1) |

### A.1.3 Concerns Identified by Identifier Analysis

| ID | Concern name |
|---|---|
| IA01 | Activation |
| IA03 | Command Execution |
| IA04 | Producer-Consumer |
| IA05 | Event Handling |
| IA06 | Undo |
| IA08 | Visitor |
| IA09 | File Handling |
| IA10 | Handling Mouse Events |
| IB01 | Area Tracking |
| IB02 | Background Drawing |
| IB03 | Managing Display Boxes |
| IB04 | Chopping Figures |
| IB05 | Clearing Figures |
| IB06 | Finding Figures |
| IB07 | Color Choosing |
| IB08 | Connecting Figures |
| IB09 | Constraining Points |
| IB10 | Figure Inclusion |

| | |
|---|---|
| IB11 | Menu Handling |
| IB12 | Managing Views |
| IB13 | Font Handling |
| IB14 | Image Handling |
| IB16 | Create Drawings |
| IB17 | Desktop Management |
| IB20 | Invocation |
| IB21 | Manipulating Figure Handles |
| IB22 | Finding Connections |
| IB23 | Dealing With Selections |
| IB24 | Inserting Figures |
| IB25 | Layout Calculation |
| IB26 | Locating Figures |
| IB27 | Moving Figures |
| IB28 | Handle and Figure Enumeration |
| IB29 | Resource Management |
| IC01 | Performing Actions |
| IC02 | Iterating Over Collections |
| IC03 | Working With Maps |

## A.2 Concerns in Tomcat

### A.2.1 Concerns Identified by Fan-in Analysis

| **ID** | **Concern name** |
|---|---|
| B1 | Consistent behavior (1) (+ Command (1)) |
| B2 | Consistent behavior (2)(Redirector - Facade) |
| B3 | Consistent behavior (3) |
| B4 | Consistent behavior (4) - Consistent mechanism for authentication (security) |
| B5 | Consistent behavior (5) |
| B6 | Consistent behavior (6) |
| B7 | Consistent behavior (7) |
| B8 | Consistent behavior (8) |
| B9 | Consistent behavior (9) |
| B10 | Consistent behavior (10) |
| B11 | Consistent behavior (11) |
| B12 | Consistent behavior (12) (+ Visitor) |
| B13 | Consistent behavior (13) |
| B14 | Consistent behavior (14) (+Visitor) |
| B15 | Consistent behavior (15) |
| B16 | Consistent behavior (16) |

| | |
|---|---|
| B17 | Consistent behavior (17) (+ Visitor) |
| B18 | Consistent behavior (18) - Security check |
| B19 | Consistent behavior (19) |
| B20 | Consistent behavior (20) |
| B21 | Consistent behavior (21) |
| B22 | Consistent behavior (22) |
| B23 | Consistent behavior (23) |
| B24 | Consistent behavior (24) |
| B25 | Consistent behavior (25) |
| C1 | Composite(1) |
| C2 | Composite (2) |
| CE1 | Contract enforcement(1) + Command(1) |
| CE2 | Contract enforcement (2) |
| CE2 | Contract enforcement(2) |
| CE3 | Contract enforcement (3) (+ Redirector-Facade) |
| CE4 | Contract enforcement (4) |
| CE5 | Contract enforcement (5) |
| CE6 | Contract enforcement (6) |
| CR1 | Chain of responsibility (1) |
| CR2 | Chain of responsibility (2) |
| EW1 | Exception wrapping (1) |
| EW2 | Exception wrappping (2) |
| LC | Lifecycle (1) |
| LO | Logging |
| O1 | Observer(1) |
| O2 | Observer(2) |
| R1 | Redirector (1) |
| R2 | Redirector (2) |
| R3 | Redirector (3) |
| R4 | Redirector (4) |
| V | Visitor |

# Bibliography


[1] Bram Adams, Kris De Schutter, Andy Zaidman, Serge Demeyer, Herman Tromp, and Wolfgang De Meuter. Using aspect orientation in legacy environments for reverse engineering using dynamic analysis — an industrial experience report. *Journal of Systems and Software*, 82(4):668–684, 2009.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM.

[3] John Anvik and Gail C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the 4th international workshop on Mining Software Repositories*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Elisa Baniassad, Paul C. Clements, Joao Araujo, Ana Moreira, Awais Rashid, and Bedir Tekinerdogan. Discovering early aspects. *IEEE Software*, 23(1):61–70, 2006.

[5] Keith Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[6] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated Software Engineering*, pages 310–315, Washington, DC, USA, 2004. IEEE Computer Society.

[7] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *ASE '06: Proceedings of the 21st IEEE/ACM international conference on Automated Software Engineering*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Silvia Breu, Thomas Zimmermann, and Christian Lindig. Mining Eclipse for cross-cutting concerns. In *MSR '06: Proceedings of the 3rd international workshop on Mining Software Repositories*, pages 94–97, New York, NY, USA, 2006. ACM.

[9] 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.

[10] Gerardo Canfora and Luigi Cerulo. How crosscutting concerns evolve in JHotDraw. In *STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 65–73, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. On the use of line co-change for identifying crosscutting concern code. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 213–222, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Mariano Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwé. A qualitative comparison of three aspect mining techniques. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Mariano Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Control*, 14(3):209–231, 2006.

[14] Grigoreta Sofia Cojocar and Gabriela Şerban. On some criteria for comparing aspect mining techniques. In *LATE '07: Proceedings of the 3rd workshop on Linking Aspect Technology and Evolution*, page 7, New York, NY, USA, 2007. ACM.

[15] Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.

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

[17] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Bernhard Ganter, Gerd Stumme, and Rudolf Wille. *Formal concept analysis: Foundations and applications (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

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

[20] L.P. Hattori and M. Lanza. On the nature of commits. In *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 63–71, Sept. 2008.

[21] 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 *Lecture Notes in Computer Science*, pages 145–164. Springer Berlin / Heidelberg, 2007.

[22] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[23] 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 4th international workshop on Mining Software Repositories*, page 30, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Zeeger Lubsen, Andy Zaidman, and Martin Pintzger. Studying co-evolution of production & test code using association rule mining. In *MSR '09: Proceedings of the 6th Working Conference on Mining Software Repositories*, pages 151–154. IEEE Computer Society, 2009.

[25] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[26] Marius Marin, Leon Moonen, and Arie van Deursen. A classification of crosscutting concerns. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 673–676, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Marius Marin, Leon Moonen, and Arie van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 29–38, Washington, DC, USA, 2006. IEEE Computer Society.

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

[29] Kim Mens, Andy Kellens, and Jens Krinke. Pitfalls in aspect mining. In *WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering*, pages 113–122, Washington, DC, USA, 2008. IEEE Computer Society.

[30] Osamu Mizuno, Shiro Ikami, Shuya Nakaichi, and Tohru Kikuno. Spam filter based approach for finding fault-prone software modules. In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, page 4, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Harold Ossher and Peri L. Tarr. Operation-level composition: A case in (join) point. In *ECOOP '98: Proceedings of the 3rd workshop on Aspect-Oriented Programming*, pages 406–409, London, UK, 1998. Springer-Verlag.

[32] Lucas D. Panjer. Predicting Eclipse bug lifetimes. In *MSR '07: Proceedings of the 4th international workshop on Mining Software Repositories*, page 29, Washington, DC, USA, 2007. IEEE Computer Society.

[33] David Shepherd, Emily Gibson, and Lori Pollock. Design and evaluation of an automated aspect mining tool. In *Software Engineering Research and Practice*, pages 601–607. CSREA Press, 2004.

[34] David Shepherd and Lori Pollock. Interfaces, aspects and views. In *LATE '05: Linking Aspect Technology and Evolution Workshop*, 2005.

[35] David Shepherd, Lori Pollock, and Tom Tourwé. Using language clues to discover crosscutting concerns. *SIGSOFT Software Engineering Notes*, 30(4):1–6, 2005.

[36] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 112–121, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Tom Tourwé and Kim Mens. Mining aspectual views using formal concept analysis. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 97–106, Washington, DC, USA, 2004. IEEE Computer Society.

[38] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. LCM: An efficient algorithm for enumerating frequent closed item sets. In *ICDM '03: Proceedings of the 3rd IEEE International Conference on Data Mining*, 2003.

[39] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *ICDM '04: Proceedings of the 4th IEEE International Conference on Data Mining*, 2004.

[40] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. LCM ver. 3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. In *OSDM '05: Proceedings of the 1st international workshop on Open Source Data Mining*, pages 77–86, New York, NY, USA, 2005. ACM.

[41] Liguo Yu and Srini Ramaswamy. Mining CVS repositories to understand open-source project developer roles. In *MSR '07: Proceedings of the 4th international workshop on Mining Software Repositories*, page 8, Washington, DC, USA, 2007. IEEE Computer Society.

[42] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production and test code. In *ICST '08: Proceedings of the 1st International Conference on Software Testing*, pages 220–229. IEEE Computer Society, 2008.

[43] Charles Zhang and Hans-Arno Jacobsen. PRISM is research in aspect mining. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 20–21, New York, NY, USA, 2004. ACM.

[44] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *MSR '04: Proceedings of the 1st international workshop on Mining Software Repositories*, pages 2–6, Edinburgh, UK, 2004.

[45] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.