Technical Report

UCAM-CL-TR-838
ISSN 1476-2986

Number 838

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Mining and tracking in evolving software

Silvia Breu

June 2013

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 2011 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Newnham College.

ISSN 1476-2986

Abstract

Every large program contains a small fraction of functionality that resists clean encapsulation. Code for, e.g., debugging or locking is hard to keep hidden using object-oriented mechanisms alone. This problem gave rise to aspect-oriented programming: such cross-cutting functionality is factored out into so-called aspects and these are woven back into mainline code during compilation. However, for existing software systems to benefit from AOP, the cross-cutting concerns must be identified first (*aspect mining*) before the system can be re-factored into an aspect-oriented design.

This thesis on mining and tracking cross-cutting concerns makes three contributions: firstly, it presents aspect mining as both a theoretical idea and a practical and scalable application. By analysing where developers add code to a program, our history-based aspect mining (HAM) identifies and ranks cross-cutting concerns. Its effectiveness and high precision was evaluated using industrial-sized open-source projects such as Eclipse.

Secondly, the thesis takes the work on software evolution one step further. Knowledge about a concern's implementation can become invalid as the system evolves. We address this problem by defining structural and textual patterns among the elements identified as relevant to a concern's implementation. The inferred patterns are documented as rules that describe a concern in a formal (intensional) rather than a merely textual (extensional) manner. These rules can then be used to track an evolving concern's implementation in conjunction with the development history.

Finally, we implemented this technique for Java in an Eclipse plug-in called ISIS4J and evaluated it using a number of concerns. For that we again used the development history of an open-source project. The evaluation shows not only the effectiveness of our approach, but also to what extent our approach supports the tracking of a concern's implementation despite, e.g., program code extensions or refactorings.

Acknowledgements

First I would like to thank my supervisor Prof. Alan Mycroft, who has provided a striking balance between freedom, trust, and guidance. His belief in me and the many discussions have been an endless source of inspiration and motivation.

I am grateful to Gates Cambridge whose scholarship has enabled me to come to Cambridge for my PhD degree, and whose fellow scholars have provided an inspiring community to be part of.

I would like to thank my fiancé Christian Lindig, for his never ending support of my endeavours away from home, and the freedom I had to go and study in one of the most exciting, rewarding, and stimulating environments. His parents' termly care packages were always a welcome surprise and provided extra energy.

Additional mention should go to Thomas Zimmermann who has been a great person to collaborate with, but also shared my love for films whenever we had an opportunity. A special thanks to Oliver Thomas who proofread this dissertation; it certainly is not a classicist's everyday's read. And last but not least a thank you is also due to the examiners of this thesis, David Greaves and Mark Harman. Their comments provided polish for this thesis.

The Cambridge University Volleyball Club and Cambridge University Women's Boat Club have not just provided plenty of experiences, competition, as well as a balance to the academic life, but also lots of wonderful friendships and unforgettable memories of companionship. Especially my Boat Race year has given me a lot of energy and oddly enough made sure I finished on time.

Dedicated to my mother.

Contents

List of Figures

List of Tables

"I have no special talent. I am only passionately curious."

Albert Einstein

1 Introduction

Building large-scale software systems is hard, but experience shows that building large-scale software systems that actually work is even harder. A famous example is the ongoing NHS IT project, which includes the implementation of the Patient Administration System. With software systems becoming more and more complex, developers face increasing difficulties in building systems in a modular fashion; this can no longer be tackled by "traditional" design and programming techniques. Once can only anticipate change successfully if the complexity of successive software releases is controlled and "code tangling" is limited.

The most commonly-used technique in industry since the early 1990s is *object-oriented programming.* This seems to be the best pragmatic choice, featuring top-down design (how the system is to be built taking into account any technical or environmental constraints) and encapsulation (concealing functional details in code units called classes). Still, every large system contains a small fraction of functionality resisting clean encapsulation, also called *cross-cutting concerns* (CCCs); if a different design was followed to get these parts well-structured, other parts would stop being well-designed—a bit like flattening orange peel. This is also referred to as "the tyranny of dominant decomposition" [48].

Taking notice of this problem gave rise to *aspect-oriented programming* (AOP [30]) as a solution: cross-cutting functionality is factored out into so-called *aspects* and these are woven back into mainline code during compilation. Using aspects makes even large-scale software systems easier to understand and modify. However, for existing software systems to benefit from aspects, the cross-cutting concerns must be identified first before the system can be re-factored into an aspect-oriented design. To again take on the picture of flattening orange peel—it is a bit like finding minimum tears in the peel. This identification task is called *aspect mining.*

This thesis, by presenting approaches to mine for cross-cutting concerns, aims at the greater vision of continously supporting developers over the development process, to make large-scale systems work better, and to avoid problems and failures such as the computerisation of the NHS. It thus addresses questions such as how aspect mining techniques in particular, and program analysis techniques in general, can support different tasks of software engineering: refactoring, program understanding, and evolution of software.

Previous approaches to aspect mining applied static or dynamic program analysis techniques to a single version of a system. While this might appear natural, we argue that such aspect mining has difficulties in being precise and in scaling to large systems. While dynamic analysis strongly depends on a compilable and executable program version as well as on the coverage of the used test cases of a program, static analysis techniques often produce too many details and false positives as they cannot weed out non-executable paths. To overcome these limitations each approach would need additional costly processing which in turn makes them less practical and less cheap. Besides, many approaches require user interaction or even previous knowledge about the program. We give a brief discussion of existing aspect mining techniques in Chapter 2.

We show that instead of looking at a single version of a system, we can leverage a system's version history to mine aspect candidates. The technique is based on the hypothesis that CCCs evolve within a project over time. A code change is likely to have introduced such a concern if the modification gets introduced at various locations within a single code change.

Our hypothesis is supported by the following example. On November 10, 2004, Silenio Quarti committed code changes "76595 (new lock)" to the EclipseCVS repository. Bug #76595 "Hang in gfk_pixbuf_new" that reported a deadlock[1] and required the implementation of a new locking mechanism for several platforms got fixed. The extent of the modification was enormous: He modified 2 573 methods and inserted in 1 284 methods a call to `lock`, as well as a call to `unlock`. As it turns out, AOP could have been used to weave in this locking mechanism at a much lower cost.

Our approach searches such cross-cutting changes in the history of a program in order to identify aspect candidates. For Silenio Quarti's changes, we find two *simple aspect candidates* $(\{\texttt{lock}\}, L_1)$ and $(\{\texttt{unlock}\}, L_2)$ where L_1 and L_2 are sets that contain the 1 284 methods where `lock` and `unlock` have been inserted, respectively. It turns out that $L_1 = L_2$, hence, we combine the two simple aspect candidates into a *complex aspect candidate* $(\{\texttt{lock}, \texttt{unlock}\}, L_1)$.

This novel approach is called *history-based aspect mining*, and collects and analyses aforementioned changes from one version to the next. Therefore it is independent of the total size of a system. And since it is static as we look at the program code that constitutes the system at certain points in its development history, it does not rely on test cases but guarantees complete coverage. It turns out that one advantage here is that we are able to mine industrial-sized software systems like Eclipse containing millions of lines of code.

In addition to the achieved scalability, this thesis explores the existing connection to a mathematical foundation called *formal concept analysis* (FCA [22]); this is an algebraic theory for specifically those *binary relations* that associate objects with attributes. Every binary relation induces a lattice of so-called *concepts* where a con-

[1] `https://bugs.eclipse.org/`

cept is a cluster of objects that have the same set of attributes. Using this connection, history-based aspect mining can then be formalised because it turns out that an aspect candidate in our approach is nothing but a cluster of objects with the same attributes within a software system. Thus, FCA can be used to efficiently [33] and correctly mine the version archives of programs in order to identify cross-cutting concerns.

However, history-based aspect mining not only identifies cross-cutting concerns and helps to improve the design of software, but also provides a completely new view on the evolution of cross-cutting concerns. So, monitoring the evolution of systems can even be further enriched—by guiding developers automatically when they change the program code for later software releases but do not update the documentation.

As part of the evolution of software systems, the effort of locating cross-cutting concerns either through manual or automatic location activities (e.g., aspect mining) can become invalidated when the system evolves. This problem can be mitigated if there is an automatic way of tracking evolving software through multiple versions, as it is very hard to read and understand tens of thousands of lines of code.

We try to achieve such automatic tracking by inferring structural patterns between the code elements that form a cross-cutting concern, and finding an intensional representation for those so that they can be used to automatically track evolving cross-cutting concerns (rather than the extensional representation of merely listing the code elements). The different patterns are based on the most common *structural relationships* found in program code (e.g., between method calls, or based on naming conventions for methods). Furthermore, there is a connection to the mathematical notion of a relation, which helps to formalise the approach, including a notion of completeness of patterns which is necessary in order to infer ample, stringent enough patterns to automatically track evolving software.

The following summarises the contents of this thesis' remaining chapters and their contributions. Chapter 2 gives an overview of related work in the area of aspect mining. It shows where the motivation for choosing a completely new point of view on the problem of identifying cross-cutting concerns comes from. Chapter 3 explains the underlying reasoning and definitions for our historical approach. Aspect candidates, simple and complex, are defined, and connections between mined patterns are established. Together they enable us to introduce the version-history-based aspect-mining approach. Next, we present the prototype implementation of HAM which was used for evaluating our aspect-mining approach in Chapter 4. There we show that our approach scales very well to industrial-sized systems, and is the first that can deal effectively with large software systems. We also discuss and compare the precision for different case studies. The sheer amount of data that needs to be processed for large-scale software requires a more efficient and smarter analysis. This gave rise to the idea of using formal concept analysis to mine for cross-cutting concerns. In Chapter 5 we show the connections of cross-cutting concerns with objects and their attributes, and how this translates to a concept lattice.

Once cross-cutting concerns are identified, we know that the acquired knowledge is at risk of being lost during subsequent code modifications. Thus, in the following chapters, we investigate whether the knowledge could be preserved by using relationships between program elements rather than keeping mere lists of program elements' names. Chapter 6 discusses extensional and intensional concern representation and introduces the various kinds of intensions we define and their inference. Chapter 7 details the implementation of our prototype tool and evaluates our approach's effectiveness in inferring implicit structure as well as in automatically tracking cross-cutting concerns' implementation. Chapter 8 provides an outlook for potential future work and concludes the thesis.

"Knowledge of what is does not open the door directly to what should be."

Albert Einstein

2 Related Work

2.1 Aspect Mining

Previous approaches to aspect mining considered a program only at a particular time, using traditional static and dynamic program analysis techniques. One fundamental problem is their scalability. While dynamic analysis strongly depends on a compiled, executable program version and on the coverage of the used program test cases, static analyses often produce too many details and false positives as they cannot weed out non-executable code. To overcome these limitations, each approach would need additional methods which in turn make them then far less practical. Besides, many approaches require user interaction or even previous knowledge about the program.

The Aspect Browser [25] was one of the first aspect mining tools introduced, and it identifies crosscutting concerns with textual-pattern matching (much like the Unix utility "grep") and highlights them in colour. The working assumption is that aspects have a signature which can be identified by a textual regular expression. This means that success or failure of the tool heavily depends on programmers having following naming conventions during development of the software system under analysis. The visual representation of the results in colour, however, is very useful and makes for easy understanding of how a concern is scattered throughout the code base being analysed.

AMT (Aspect Mining Tool) [27] is based on a multi-modal analysis to avoid the weakness of each individual analysis by using the strength of the toher. While text-based analysis is language independent, it does suffer as mentioned above from heavily depending on same types where strict naming conventions have been followed. AMT therefore combines text- and type-based analysis of source code to reduce false positives, since type-based analysis works better for objects of different types but similarly names. A later extension of the tool, called AMTex by [54], tries to overcome the limitation of visualisation-based aspect mining as well as dealing with industrial-sized software systems. It offers more analytical functionality than the original AMT such as composition of mining activities, manages mining tasks, and cross-analyses the mining results.

The exploration tool JQuery [28] is implemented as Eclipse plugin and provides a generic browser that allows logic queries to be defined in a tool specific query language. The queries can be run on an abstract representation of a Java program's source code, after which the results are displayed in an initial browser view. The analysis of the source code by navigation can be based on structural relationships, regular expression matches, or complex searches for structural patterns. Navigating the tree in the browser view, the user can extend the initial results by making additional queries of which the results are added as subtrees with the elements linked to the source code for easy further investigative navigation. This has the advantage that the user does not have to switch between view or queries which helps the developer to stay focussed and not lose orientation when exploring the identified crosscutting concerns.

FEAT [42] (Feature Exploration and Analysis Tool) is an aspect mining (and general software exploration) tool and implemented as Eclipse plugin. It visualises concerns in a system using so-called *concern graphs*. A concern graph abstracts a concern's implementation details by storing the structure implementing that concern, e.g., that a method reads a specific field, or creates an object of a certain class. Thus it documents explicitly the relationships between the different concern elements. A concern graph's nodes represent classes, fields, and methods; its edges represent different kinds of relations between the nodes. FEAT is query-based, with the query process being interactive: the user can submit additional queries based on the results. However, this requires a starting point for the analysis, which sometimes has to be found by trial and error. The difficulty to find such a point increases if the software system to be analysed is unknown or large, both quite common and realistic situations. This means that domain knowledge of the software system to be analysed has to be acquired if not present before aspect mining using FEAT is possible.

Another framework for automated aspect mining, Ophir [45], uses a control-based comparison inspired by code clone detection. It discovers initial candidates for refactoring using program dependence graphs (PDG). In the next step, undesirable refactoring candidates undergo data-based filtering, looking for similar data dependencies in PDG subgraphs representing code clones. The last phase of the approach is the so-called coalesce phase; it identifies similar candidates and coalesces these pairs into sets of similar candidates, which are the final set of refactoring candidate classes.

Tourwé and Mens [50] introduce an identifier analysis based on formal concept analysis for mining *aspectual views*. An aspectual view is a set of source code elements, e.g., class hierarchies, classes, and methods, that are structurally related. Their mining approach has five phases: generate elements and properties, generate the concept lattice, filter out unimportant concepts, analyse and classify remaining concepts, and finally display concepts. The proposed identifier analysis discovers interesting and meaningful aspectual views, producing many details, however, due to the lightweight technique used, it suffers from many false positives despite filtering out unimportant concepts in phase three of the analysis. False negatives can also occur due to program elements not sharing the exact same substring, thus splitting one concept in two or more. As with other approaches discussed, naming conventions not followed can aggravate this problem. However, the results are meaningful and interesting from a program understanding point of view.

Krinke and I [32] propose an automatic static aspect mining based on control flow. First, the control flow graph of the program under analysis is computed. The graph then gets traversed, extracting uniform and crosscutting execution relationships. An initial evaluation showed that identified crosscutting concern candidates are not concerns that can be refactored following the AO paradigm, or the detected crosscutting concern is perfectly good style. However, the results do provide interesting insights into the crosscutting behaviour of the analysed program and thus are valuable for program understanding. In addition, the technique can be used to identify crosscutting anomalies in discovered execution relation patterns.

The fan-in analysis by Marin, van Deursen, and Moonen [38] determines methods that are called from many different places, thus having a high fan-in, and in consequence representing functionality that is used across the software system. The analysis consists of three major steps. First, the fan-in metric for all methods in the source code being analysed is computed, based on which they are then sorted so as to aid inspecting methods with high fan-in. In the second step, the results are filtered, e.g., by requiring a certain threshold for the fan-in value, or eliminating getters and setters. In the last step, which is mostly done manually, the remaining methods and their callers and call sites are inspected. This approach proved to be able to detect crosscutting concerns, though aspects that only leave a small 'footprint' in the source code (thus have a lower fan-in value) are missed. Our approach is similar in that we analyse how fan-in changes over time.

Another aspect mining approach, DynAMiT (Dynamic Aspect Mining Tool) [4, 7], analyses program traces reflecting the run-time behaviour of a system rather than its static structure. The approach is inspired by the shopping basket analysis approach in data mining. Different program executions generate program traces to build the data pool used for aspect mining. The traces are then investigated for recurring execution patterns based on different constraints, e.g., the requirement that the patterns have to exist more than once or in different calling contexts in the program trace. Breu also reports on a hybrid approach [6] where the dynamic information of the previous DynAMiT approach is complemented with static type information such as static object types in order to remove ambiguities and thus improve on the results by reducing false positives. Additionally, considering method calls rather than method executions in the traces finds more aspect candidates.

Tonella and Ceccato [49] suggest a dynamic code analysis technique. First, execution traces for the main functionality of a software system are generated. Then, formal concept analysis is applied to the relationship between execution traces and executed computational units (i.e., class methods). In the resulting lattice, aspect candidates are detected by determining class methods that come from multiple modules (i.e., classes) and belong to multiple use-cases. The approach is semi-automated. If no use cases for execution trace generation exist, these have to be defined. After that everything is automated up to computing the concept lattice. Interpreting the lattice and identifying concerns for refactoring into aspects is then manual again. No prior knowledge of the software is needed for this interpretation step though, which is an advantage over some of the other techniques. As with all dynamic analysis approaches, its main limitation is incompleteness, but also includes the quality of the result being dependent on the right granularity of the use cases.

Loughran and Rashid [35] addressed the question for the best representation of aspects found in a legacy sytem as to find a way to provide the best tool support for aspect mining. Two general approaches were considered. Direct aspect storage combined with meta-data, and mapping of aspect anatomy to the database model. The conducted evaluation revealed that the first approach is easy to implement as well as fast and efficient. The second approach has very different but equally important strengths: it retains the aspect representation and enables fine-grained aspect mining. Thus, the authors suggest as probably best solution a hybrid approach combining both, direct aspect storage coupled with meta-data together with mapping aspect anatomy to the database model.

Binkley et al. [2] have looked into a semi-automatic approach to help refactor object-oriented code into aspect-oriented code. Their technique assumes that prior to applying it, aspect mining has been performed that as a result produces code where markers indicate the beginning and end of those fragments that can be aspectised. Their argument for a humanly guided process is that while a certain amount of automation is possible and sensible, decisions regarding tradeoffs and how to migrate semantics preserving, without changing functionality and behaviour of the software, require the specialist knowledge of developers. Binkley et al.'s technique consists of four steps called *discovery* (determine applicable refactorings), *transformation* (apply OO transformations), *selection* (select refactorings), and *refactoring* (transform code). These steps are iterated until a fixed point is reached, i.e., until all identified concerns hat were marked as aspectisable have been aspectised.

In [3] they extend their work on this by introducing more refactorings that support the migration from the object-oriented to the aspect-oriented paradigm, reimplement their original prototype as an Eclipse plugin, and evaluate the effectiveness of their approach in an extensive case study involving four medium-sized software systems. They conclude that it is possible to migrate code from the OO to the AO paradigm largely automatically, though the identified refactorings cannot be done entirely automatically but require well-understood transformations such as, e.g., side effect removal; however, these transformations can again be automated. However, they also argue that it is neither desirable nor achievable though to automate the whole process completely as in some cases a human expert is best suited to make refactoring decisions.

Code clones have been connected with crosscutting concerns for a while, suggesting that they could be refactored into aspects if the usual object-oriented paradigm is not sufficient. Bruntink et al. [10] have evaluated how capable clone detection techniques can be for identifying crosscutting concerns automatically with the long-term vision of developing an automatic concern miner based on their findings. For their experiment, they chose an industrial software component, in which an expert developer manually marked occurrences of four types of crosscutting concerns, namely error handling, tracing, parameter checking, and memory management. Further, they chose two different clone detection tools, one from the category of AST-based clone detectors, and one based on tokenised source code representation. The study revealed that certain crosscutting concerns such as parameter checking and memory error handling can be identified very well using clone detection techniques, while tracing concerns and error handling prove tricky.

One of the most frequently used techniques for mining version archives is co-change. The basic idea is simple: *Two items that are changed together in the same transaction (program-update) are related to each other.* These items can be of any granularity; in the past co-change has been applied to changes in modules, files, classes, and methods. Our approach is also based on co-change. However, we use a different, more specific notion of co-change. Methods are part of a (simple) aspect candidate when they are changed together in the same transaction and *additionally the changes are the same*, i.e., a call to the same method is inserted.

Recently, research extended the idea of co-change to *additions* and applied this concept to method calls: *Two method calls that are inserted together in the same transaction are related to each other.* Williams and Hollingsworth use this observation to mine pairs of functions that form usage patterns from version archives [52]. By doing so, they recover system-specific rules that describe, for example, how functions interact in the source code, but that have been either left undocumented or changed over time along with the changes in the source code. Their tool analyses each version of a file in the repository and identify new function usage patterns that got introduced in subsequent versions of each file. However, this also creates huge amounts of data, which, with growing project size, will require filter mechanisms. Also, removed patterns are currently not tracked, thus stay in the list of patterns in a software project even after they cease to exist.

Livshits and Zimmermann use data mining to locate patterns of arbitrary size and apply dynamic analysis to validate their patterns and identify violations [34]. Their tool DynaMine, implemented as Eclipse plugin, pre-processes revision history to find method calls that were inserted and stores them in a database. Then, this database is mined for usage and error patterns. The user can then select patterns and run the accordingly instrumented program, collecting dynamic data. Any pattern violations are then presented to the user in Eclipse. The tool allows for interaction as well: the user can go back, change the patterns, and re-instrument the program, rerunning the dynamic analysis. Our work also investigates the addition of method calls. However, within a transaction, we do not focus on calls that are inserted together, but on locations where the same call is inserted. This allows us to identify cross-cutting concerns rather than usage patterns.

2.2 Concern Tracking

These approaches discussed so far can all help to locate and document concerns in source code. However, as mentioned before, the next step after mining for concerns is actually to track and update that information as software systems evolve. Thus, both activities, mining and tracking, have closely related goals. In the remainder of this chapter we will highlight known techniques.

A number of automated and semi-automated approaches have been proposed to help developers map high-level features to code entities. Feature location complements our approach because it can be used to provide the initial concern mappings to ISIS4J. Feature location approaches often join multiple analysis techniques. One such example is [55] where information retrieval and branched call graphs are combined. The approach contains four major steps: acquire specific connections between features and functions (using information retrieval), choose initial specific function for each feature, then determine relevant function and pseudo execution traces using the branched call graph extracted from the source code, and determine the final specific functions. The evaluation of the technique generated too many irrelevant pseudo execution traces but showed promising results in the other steps, in particular for identifying specific functions and relevant functions.

[40] combine probabilistic ranking and latent semantic indexing to identify features in source code. Both techniques produce a set of ranked facts from the software system to be analysed as result to the feature identification. The probabilistic ranking ranks events observed while executing the software under given scenarios; the second technique is an information retrieval task using latent semantic indexing of the source code implementing the software system being analysed. The authors show in their case studies that combining both techniques increases the precision of feature identification and combined they perform better than either applied alone. Furthermore, this novel combination of two techniques works well with large systems such as Mozilla.

Other approaches use dynamic analysis to find code entities that were used by certain features. One such technique is used in STRADA [16], an Eclipse-based tool that captures and analyses *scenario-based* execution traces in order to ultimately recover traceability links between development artifacts and features or requirements [17]. The problem addressed here is the challenge of developers to understand the relationship between a software's source code and its requirements; also referred to as trace links. Creating and especially maintaining trace links is labour intensive and a process prone to errors. STRADA (*S*cenario-based *TRA*ce *D*etection and *A*nalysis) enables developers to investigate trace links through testing and can be applied to any software system that is observable during executiong, meaning source code can be linked to requirements and features by executing it during testing, thus automatically creating traceability.

Eisenbarth, Koschke et al. [19, 31] combine dynamic and static analyses to locate features in source code. Their approach is of semi-automatic nature and aims at identifying the computational units implementing a feature and the set of required computational units for a feature. In order to achieve this, they propose a five-step process: scenario creation by domain expert based on features, extraction of static dependency graph, dynamic analysis by using the software system to be analysed following scenarios created in the first step, application of formal concept analysis and interpretation of concept lattice to identify relevant computational units, and static dependency analysis to identify any additional computational units relevant to selected features. One major advantage of this approach is its ability to handle scenarios that invoke several features. Additionally, the proposed technique allows for incremental exploration of features.

The idea of explicitly representing the scattered code implementing a concern goes back to Soloway et al. [46], who looked at how the software process could be facilitated.

They showed that identifying crosscutting code is time-consuming and error-prone. . They found that conceptual representation methods have a positive impact on the software development process, and propose the writing of cross-referenced textual documentation in the form of annotations to code.

Numerous approaches address the problem of documenting and representing concerns *intensionally*, meaning by different means than listing names of program elements comprising a concern. With FEAT [43], the Feature Exploration and Analysis Tool described already more detailed in the previous section of the chapter, a user can create and visualise *concern graphs* in the Eclipse development environment by selecting relationships similar to ISIS4J intensions (e.g. all methods accessing a certain field). through the graphical user interface instead of writing queries.

An example of a query approach is the JQuery [28] exploration tool also detailed earlier in the chapter when discussing aspect mining techniques and tools. It provides a language similar to Prolog, allowing a user to select Java elements. Concerns can be described using different structural characteristics (e.g. hierarchy relationships, method calls, field accesses, etc.) and regular expression matches.

The previously discussed Aspect Browser[25], besides being useful for aspect mining, also can be seen as a tool to document concerns, despite the simple underlying idea of identifying crosscutting concerns with textual-pattern matching followed by highlighting them in the source code browser.

A different usage of queries was proposed by Mens and Kellens [39] with IntensiVE, the Intensional View Environment, a tool suite aimed at preventing the qualitative deterioration of a software system's implementation regarding design structures, coding conventions and other so-called structural regularities. If the quality is compromised or requirements are violated, IntensiVE will report this to avoid the software system becoming so crippled that any further changes become time-consuming and costly. In particular, users can define views as a set of structurally similar classes and methods in a Smalltalk program. This is done according to logical rules that resemble ISIS4J intensions (e.g., all classes that declare a method 'accept' with a single parameter), thus allowing to check for example specific architectural rules that are otherwise only specified in accompanying documents that notoriously become out of date, or might not even be available as it is all 'in the heads' of the developing team.

Aspect-Oriented Programming (AOP) [29] can also be used to document *crosscutting* concerns intensionally. Pointcut languages, e.g., AspectJ [1], define naming and runtime patterns that ultimately describe the code elements where aspects will be injected, meaning (additional) code that should be executed. As the name suggests, such languages use pointcuts to achieve this. A pointcut is a predicate (in the form of specifically defined criteria) that matches join points, where a join point refers to a point during the execution of a program. The mentioned code that should be executed at all join points matching a specific pointcut is also called *advice*.

ConcernMapper [RWW05] is a concern management tool for Java software projects, implemented as an Eclipse plugin. It allows the developer to form a collection of program elements that are concerned with a concern's implementation, and save it for future use or reference. It thus enables reorganising the modularity of a softare system suited to your needs with changing or touching the actual source code. The concern representation is created simply by dragging and dropping the program elements of interest into a view. This kind of concern representation is of the extensional kind in contrast to the earlier mentioned intensional forms of representing features and concerns. We will use this tool for our own ISIS4J approach introduced in this thesis in later chapters.

CME [11], the Concern Manipulation Environment, is a suite of tools that help create, manipulate, and evolve aspect-oreinted software systems. It provides a common platform for different tools to integrate and interoperate, as well as frameworks that offer language- and paradigm-independent access to low-level concern representation and manipulation capabilities. One of the frameworks is the Pattern Matcher called PUMA [47]. It is a generic query engine which included a proposal to enable users to define a concern intension using any kind of query languages that could potentially select any kind of artifacts (code elements, ant tasks,[2] documents, etc.).

All of these approaches allow users to document concerns in a way that can be robust in the face of evolving software by relying on intensions. Although our technique also uses intensions to track concerns in evolving source code, the main innovation of our work is to provide a system which allows the automatic generation of intensions from purely extensional descriptions, such as the ones that could be produced by a feature location technique. By minimising the cost of producing robust concern description, we hope to make their use more prevalent in the maintenance of long-lived systems.

Various techniques and tools have been developed to extract different types of implicit structure from development artifacts. Marin et al. developed an aspect mining framework that identifies crosscutting concern (CCC) sorts. These are rule sets describing certain concern types [37]. For example, if all methods accessing a database also open and close a connection, the framework will detect a "Consistent Behavior" CCC sort. This sort can then be used to find all methods exhibiting this behaviour or to visualise at a higher level the cross-cutting concerns present in the software project. This framework uses techniques such as concept analysis and fan-in analysis to perform the rules inference. As opposed to CCC sorts that must match specific rule sets, ISIS4J can come up with any rule combination and is not restricted to one kind of concern.

Ernst et al. [20] also proposed an approach to discover function invariants to support software evolution by using dynamic analysis. Their technique can discover pre- and post-conditions such as "`variable a` should be equal to the size of `array b`" when entering a function and "the elements of `array b` should not have been modified" at the end of the function. As is the case for ISIS4J, their technique benefits from fixed-point inference since detected invariants are used to detect new invariants. Although both approaches aim at supporting software evolution, ISIS4J performs its inference on coarser-grained elements and focusses on concern mapping instead of function invariants.

[2]`ant.apache.org`

"The only reason for time is so that everything doesn't happen at once."

Albert Einstein

3 History-Based Aspect Mining: Basic Technique

Previous approaches to aspect mining considered only a single version of a program using static and dynamic program analysis techniques, and thus would look at a program at only one point in time. Our approach introduces an additional dimension of time: the *history* of a project.

We model the history of a program as a sequence of transactions. A *transaction* collects all code changes between two versions, called *snapshots*, made by a programmer to complete a single development task. Technically a transaction is defined by the revision control system that is used for the code base we analyse. In our case, this is CVS. However, our approach extends to arbitrary version archives. This and the following two chapters are a reworking of the paper on Mining Aspects from Version History [8] and the paper on Mining Eclipse for Cross-Cutting Concerns [9].

3.1 Background

In software development, often several developers work within one and the same code base, and thus may be changing the same files. It is thus necessary to organise such projects and avoid repeated changes back and forth, or conflicts because one developer overwrites another developers' work. Software tools for *revision control* are thus standard in multi-developer projects.

Revision Control. *Revision control systems* (often also referred to as *version archives*) automatically manage changes to documents and source code, often from developers spread apart in space or time. Changes are usually identified by increasing numbers, with the initial set of files being named "revison 1", and after each submitted change, the newly resulting set of files being named “revision 2”, “revision 3”, and so on. Revisions also contain a timestamp of the submitted change, the name of the person who made the change, and in most cases a comment regarding why something has been changed, e.g., “corrected spelling mistakes in Chapter 3”. Revision control also enables a development team to go back to an earlier version and start working from there again. One established and well-known free software revision control system is the Concurrent Versions System (CVS) [12]. We will use the vast amount of data a revision control system can provide to mine for aspect candidates.

Data Mining and Co-Addition. *Data mining* is about systematically analysing (usually vast amounts of) data, and also the whole process of extracting patterns from such data. It has been researched and used for decades for fraud detection, surveillance tasks, profiling to detect terrorist activities, or marketing, to name just a few areas. One of the most famous examples is probably the shopping basket analysis, where, e.g., it was found that if a young man after 8 pm buys nappies, he will also buy a six-pack of beer as he is most likely a young father on babysitter duties. Such insights are widely used, for example, to analyse which products get bought together in order to, for example, re-arrange where goods are on display in a supermarket. Over the past years, data mining has also been introduced into the area of software engineering, in particular on version archives such as CVS. It is used to predict software quality, to find defects, to analyse software evolution, and for many more problems.

One of the most frequently used techniques for mining version archives is co-change. The underlying idea is simple: Two items (e.g., methods, classes or packages) that are changed together in the same transaction are related to each other. This concept has been adapted to additions of items, in particular to the addition of method calls. It is called *co-addition*: two method calls that are inserted together in the same transaction are related to each other; they have been co-added. We will use this principle as the basis of our technique to identify aspect candidates from the development history of software systems.

3.2 Basic Definitions

In previous work, we introduced dynamic and hybrid approaches for aspect mining [4, 7, 5], focussing on the analysis of execution traces of programs. This was the motivation to look at changes to programs that insert or delete method calls, as such changes, assuming that they are not dead code, also directly change execution traces. As our hypothesis is that cross-cutting concerns do not necessarily exist from the beginning but may be introduced over (development) time, we concentrate on the addition of method calls and omit deletions. For our purposes, this also simplifies a developer's transaction in a version archive to a set of inserted method calls only:

Definition 1 (Transaction)
A transaction *T is a set of pairs (m, l). Each pair (m, l) represents an insertion of a call to method m in the body of the method l.*

The method l into which a method call m is inserted is also called *method location*. To avoid ambiguities we identify l by its full signature, including package and class name. The inserted method m is identified only by its name as well as the number of arguments; this is a safe reduction that helps to reduce costs for preprocessing (see Section 4.1).

For our technique, we need to access and handle certain meta-data of a transaction T, namely the committing developer, the time of the commit, methods that were changed, and naturally the methods that were added in a transaction. We name these as follows:

developer(T) is the name of the developer who committed transaction T.

timestamp(T) is when a transaction T was committed.

locations$(T) = \{l \mid (m, l) \in T\}$ is the set of methods that were changed in transaction T.

calls$(T) = \{m \mid (m, l) \in T\}$ is the set of method calls that were added in transaction T.

Now, we are searching for so-called *aspect candidates* in the set of transactions $\mathcal{T}$. An aspect candidate represents a cross-cutting concern that could also be implemented as an aspect in the AOP sense. It consists of one or more calls to methods M which are spread across two or more method locations L. Depending on its exact form, it can either be a *simple* or a *complex* aspect candidate. It is defined as follows:

Definition 2 (Aspect Candidate)
An aspect candidate $c = (M, L)$ *consists of a non-empty set M of methods and a non-empty set L of locations where each location $l \in L$ calls each method $m \in M$. If $|M| = 1$, the aspect candidate c is called* simple*; if $|M| > 1$, it is called* complex.

This means that each method call m added in transaction T leads to a potential aspect candidate c. The function SIMPLE_CANDIDATES($\mathcal{T}$) in Algorithm 1 does exactly that: it returns for every transaction $T \in \mathcal{T}$ and every method call $m \in$ *calls*(T) one aspect candidate.

Filtering for Large Projects. As we said before, an aspect candidate is at least one method call spread across at least two locations. Furthermore, the results for projects like Eclipse that have millions of lines of code and a long history would be enormous. Therefore, we use filtering to find actual aspect candidates and decrease the number of false positives.

To find only aspect candidates that cross-cut a considerable part of a program, we will—for our purposes—ignore all candidates $c = (M, L)$ where fewer than eight locations are cross-cut, i.e., $|L| < 8$. We believe that maintenance will benefit most from refactoring such candidates into aspects as they are homogeneous and of considerable size, and thus express cross-cutting concerns. The cut-off value of eight is chosen based on previous experience [34]. This of course does not mean that the cut-off value cannot be adjusted to suit other projects' sizes.

Algorithm 1 Simple aspect candidates

```
1: function CANDIDATES(T)
2:     $C_{result} = \emptyset$
3:     for all $m \in calls(T)$ do
4:         $L = \{l \mid l \in locations(T), (m, l) \in T\}$
5:         $C_{result} = C_{result} \cup \{(\{m\}, L)\}$
6:     end for
7:     return $C_{result}$
8: end function
9:
10: function SIMPLE_CANDIDATES($\mathcal{T}$)
11:     return $\bigcup_{T \in \mathcal{T}}$ CANDIDATES($T$)
12: end function
```

Complex Aspect Candidates. However, if we look at AOP literature, then it becomes apparent that challenging cross-cutting concerns often consist of more than one method call. Remember, for example, the `lock/unlock` concern presented in Chapter 1. In order to find such concerns, we can combine two (simple) aspect candidates $c_1 = (M_1, L_1)$ and $c_2 = (M_2, L_2)$ to a *complex aspect candidate* $c' = (M', L')$ with $M' = M_1 \cup M_2$ and $L' = L_1$, if c_1 and c_2 cross-cut exactly the same locations, i.e., $L_1 = L_2$. This last condition, $L_1 = L_2$, might look restrictive, but actually it is sensible. Method calls inserted in the same locations are most likely related.

Algorithm 2 on page 28 constructs such complex aspect candidates. Function COMPLEX_CANDIDATES takes all simple aspect candidates located earlier as input and combines candidates with matching method locations into a new complex aspect candidate. It is worthwhile to note that the resulting complex aspect candidates are also combined from simple aspect candidates with different transactions as their origin.

Algorithm 2 Complex aspect candidates

```
1: function COMPLEX_CANDIDATES($C_{simple}$)
2:     $C_{result} = \emptyset$
3:     for all $(M, L) \in C_{simple}$ do
4:         $\mathcal{M} = \{M' \mid (M', L) \in C_{simple}\}$
5:         $M_{complex} = \bigcup_{M' \in \mathcal{M}} M'$
6:         $C_{result} = C_{result} \cup \{(M_{complex}, L)\}$
7:     end for
8:     return $C_{result}$
9: end function
```

3.3 Ranking Techniques

As one can probably imagine, mining in thousands of transactions, if the program is of industrial size, will produce a vast number of aspect candidates. Not all results will be as good an aspect candidate as others. What would be nice is a ranked list with the most likely or suitable aspect candidates near the top of the list. In order to achieve that, certain ranking techniques can be applied. In the following, we present three ranking techniques we defined on our located aspect candidates. They take into account different observations, from frequently-called library methods, via fragmented introduction (of aspect candidates) over time, to compact introduction of aspect candidates.

Rank by Size. The first intuitive suggestion is that aspect candidates that cross-cut numerous locations should be more interesting. After all, on the same principle, we suggested to only consider aspect candidates above a certain cut-off size, depending on the overall size of a project. Therefore, the first, simple ranking technique we propose is to sort all identified aspect candidates $c = (M, L)$ by their size $|L|$, with decreasing size as we go down the ranked list. As a result, we get those candidates first, that are the most cross-cutting. However, this still contains a certain amount of noise, depending on how extensively a project's programmers use Java library methods. For example, method calls that would be frequent but do not necessarily represent cross-cutting functionality in an AOP sense, include `iter()`, `hasNext()`, or `next()`.

Rank by Fragmentation. The second rating presented here will penalise the previously mentioned common Java method calls when their introduction in many locations is spread across many transactions. If a cross-cutting concern is added to a system and not changed later on, it appears in only one transaction. In order to capture such aspects, and boost them up to the top of the ranking, we sort aspect candidates by the number of transactions in which we find a candidate. If an aspect candidate is found in fewer transactions, it is considered to be more likely not to be a false positive or of low interest. We refer to this count of transactions, in which an aspect candidate $c = (M, L)$ appears, as the *fragmentation* of aspect candidate c:

$$\mathit{fragmentation}(c) = |\{T \in \mathcal{T} \mid M \subseteq \mathit{calls}(T)\}|$$

Should two or more aspect candidates have the same fragmentation because they were added in the same number of transactions, we rank additionally by size $|L|$, the number of locations in which the aspect candidate has been introduced. The more cross-cutting it is, the higher it will be ranked within the same fragmentation count.

Rank by Compactness. Similar to the previous ranking by fragmentation, this ranking technique has the advantage that common Java method calls are ranked low. It is safe to assume that in most cases the developer will not introduce a cross-cutting concern only partially, then do something completely different and commit to the version archive, then return to complete the rest of the original work on introducing a

CCC and commit that. But a cross-cutting concern that has been introduced in one transaction, may get extended to additional locations in later transactions for various reasons. However, such a cross-cutting concern would be ranked low under the previous ranking techniques. Thus, we introduce a third ranking technique called *compactness* in which aspect candidates with high compactness appear at the top of the list, and less compact aspect candidates are pushed to the bottom of the ranking. The *compactness* of an aspect candidate $c = (M, L)$ is the ratio between the size $|L|$ and the total number of locations where calls to M occurred together in the development history:

$$\mathit{compactness}(c) = \frac{|L|}{|\{l \mid \exists T \in \mathcal{T}, \forall m \in M \colon (m, l) \in T\}|}$$

Again, should two or more aspect candidates have the same compactness, we will apply the size $|L|$ as an additional (second) ranking criterion.

3.4 Locality and Reinforcement

In experiments, we observed that cross-cutting concerns can be introduced within one transaction and later inserted in other new locations in later transactions. We refer to this as "extending a cross-cutting concern to new locations later". This happens when a developer introduces changes but later recognises that the task was not complete with her last commit. She will then complete the introduction of a cross-cutting concern. This also can happen when a software system grows and an existing cross-cutting concern has to be extended to the newly added functionality, too. Although such concerns are recognised by our technique as multiple, different aspect candidates, these candidates may be ranked low and missed, despite the compactness criterion we introduced earlier.

To strengthen aspect candidates that were inserted in several transactions, we use the concept of *locality*. Two transactions are locally related if they were created by the same developer, were committed around the same time, or changed the same locations. If there exists locality between transactions, we mutually *reinforce* their aspect candidates.

Temporal Locality refers to the fact that aspect candidates may appear in several transactions that are close in time. In Figure 3.1 on page 31 there exists temporal locality between transaction 4 and transactions 3 and 5.

Possessional Locality refers to the fact that aspect candidates may have been created by one developer but committed in different transactions; thus they are *owned* by her. Girba et al. [23] define ownership by the last change to a line; in contrast, we look for the addition of method calls, which is more fine-grained. In Figure 3.1 there exists possessional locality between transaction 4 and transactions 1, 2, and 7, all of them were committed by Mary.

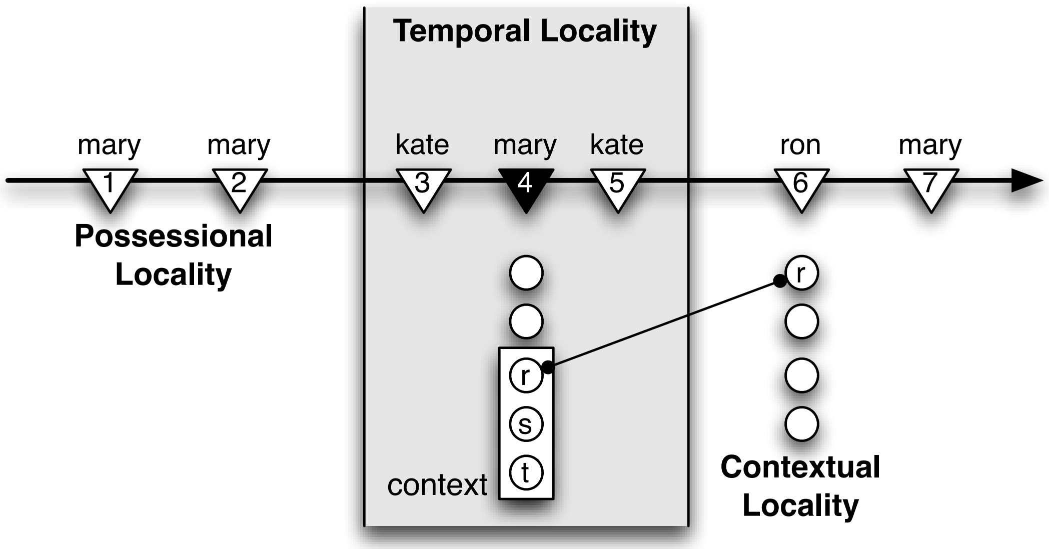


Figure 3.1: Possessional, temporal, and contextual locality for transaction 4. Transactions are denoted with $\bigtriangledown$ and changed methods as $\bigcirc$.

Spatial/Contextual Locality refers to the fact that aspect candidates may have been created in different transactions that are linked by the methods being modified. Spatial locality is caused by any method that is modified in two transactions, while contextual locality is caused by only those methods that are in a given context, for which we will use the set L of cross-cut methods of an aspect candidate $c = (M, L)$. In Figure 3.1 there exists spatial locality *and* contextual locality in the context $L = \{r, s, t\}$ between transactions 4 and 6 because transaction 6 modified location $r \in L$.

Definition 3 (Locality)
Let $T_1, T_1 \in \mathcal{T}$ be arbitrary transactions, $c = (M, L)$ be an aspect candidate, and t be a fixed time interval. We say T_1 and T_2 have

(a) temporal locality*, written as* $T_1 \overset{time}{\leftrightsquigarrow} T_2$ *iff*

$$|timestamp(T_1) - timestamp(T_2)| \leq t$$

(b) possessional locality*, written as* $T_1 \overset{dev}{\leftrightsquigarrow} T_2$ *iff*

$$developer(T_1) = developer(T_2)$$

(c) spatial locality*, denoted as* $T_x \overset{spat}{\leftrightsquigarrow} T_y$ *iff*

$$locations(T_x) \cap locations(T_y) \neq \emptyset$$

(d) contextual locality *in the context of c, denoted as* $T_x \overset{c}{\leftrightsquigarrow} T_y$ *iff*

$$c \in \textsc{Candidates}(T_x) \quad \wedge \quad L \cap locations(T_y) \neq \emptyset$$

Assume that we found two aspect candidates $c_1 = (M_1, L_1)$ and $c_2 = (M_2, L_2)$ in two different transactions where the called methods are the same, i.e., $M_1 = M_2$. If there exists locality of any form between these two transactions, we can combine both aspect candidates. As a result we get a new aspect candidate $c' = (M_1, L_1 \cup L_2)$. We call this process *reinforcement*.

Definition 4 (Reinforcement)
Let $c_1 = (M_1, L_1)$ and $c_2 = (M_2, L_2)$ be aspect candidates. If $M_1 = M_2$, the construction of a new aspect candidate $(M, L_1 \cup L_2)$ with $M = M_1 = M_2$ is called reinforcement*.*

We only reinforce if there exists locality between two transactions. After having run some experiments, however, we decided not to take spatial locality into account as it creates noise by linking unrelated transactions together and thus does not improve the accuracy of our mining results. Instead, we found a way to get a similar effect without using spatial locality: contextual locality with aspect candidates as context relates transactions in a similar way, but with considerably less noise.
We implemented four reinforcement algorithms, which are listed in Algorithm 3 on pages 33 et seq. The functions for temporal (TEMPORAL), for possessional (POSSESSIONAL), and for contextual reinforcement (REINFORCE_CONTEXTUAL) are very similar in style. They all

1. take a set $\mathcal{T}$ of transactions as input,
2. identify for each transaction T other transactions $\mathcal{T}_{loc}$ that are related to T with respect to some form of locality
3. compute simple aspect candidates for each of these transactions, and
4. build new combined, or *reinforced* candidates.

The functions TEMPORAL, POSSESSIONAL and REINFORCE_CONTEXTUAL differ in the kind of locality that is used to identify related transactions (see second step). For contextual locality (REINFORCE_CONTEXTUAL) this step is pushed into the inner **for all** loop because the related transactions depend on the chosen aspect candidate.

Additionally, we implemented an algorithm ALL that combines the results of temporal and possessional reinforcement. However, it does not use several kinds of localities at the same time as this could reinforce all transactions. If this happened, the unique historical perspective of our approach would get abandoned and many transactions over a long time could “melt” into one big transaction which is then the same as previous approaches that looked at the software system only at one point in time. Instead, the algorithm ALL applies the different kinds of localities independetly (sequentially).

Algorithm 3 Reinforcement algorithms

```
function REINFORCE($\mathcal{T}$, $x \in \{\text{time}, \text{dev}\}$)
    $C_{reinf} = \emptyset$
    for all $T \in \mathcal{T}$ do
        $\mathcal{T}_{loc} = \left\{T' \mid T' \in \mathcal{T}, T' \overset{x}{\leftrightsquigarrow} T\right\}$
        $C_{loc} = \bigcup_{T' \in \mathcal{T}_{loc}} \text{CANDIDATES}(T')$
        for all $c = (M, L) \in \text{CANDIDATES}(T)$ do
            $L_{reinf} = \{L' \mid c' = (M', L') \in C_{loc}, M' = M\}$
            $C_{reinf} = C_{reinf} \cup \{(M, L_{reinf})\}$
        end for
    end for
    return $C_{reinf}$
end function

function TEMPORAL($\mathcal{T}$)
    return REINFORCE($\mathcal{T}$, time)
end function

function POSSESSIONAL($\mathcal{T}$)
    return REINFORCE($\mathcal{T}$, dev)
end function

function REINFORCE_CONTEXTUAL($\mathcal{T}$)
    $C_{reinf} = \emptyset$
    for all $T \in \mathcal{T}$ do
        for all $c = (M, L) \in \text{CANDIDATES}(T)$ do
            $\mathcal{T}_{loc} = \left\{T' \mid T' \in \mathcal{T}, T' \overset{c}{\leftrightsquigarrow} T\right\}$
            $C_{loc} = \bigcup_{T' \in \mathcal{T}_{loc}} \text{CANDIDATES}(T')$
            $L_{reinf} = \{L' \mid c' = (M', L') \in C_{loc}, M' = M\}$
            $C_{reinf} = C_{reinf} \cup \{(M, L_{reinf})\}$
        end for
    end for
    return $C_{reinf}$
end function

function ALL($\mathcal{T}$)
    return TEMPORAL($\mathcal{T}$) $\cup$ POSSESSIONAL($\mathcal{T}$)
end function
```

"Not everything that can be counted counts, and not everything that counts can be counted."

Albert Einstein

4 Evaluation History-Based Aspect Mining

In the introduction to the thesis in Chapter 1 we told an anecdote how we identified cross-cutting concerns in the history of Eclipse (`lock/unlock`). Another example for a cross-cutting concern is the call to method `dumpPcNumber` which was inserted into 205 methods in the class `DefaultBytecodeVisitor`. This class implements a visitor for bytecode, in particular one method for each bytecode instruction; the following code shows the technique for instruction `aload_0`.

```
/**
 * @see IBytecodeVisitor#_aload_0(int)
 */
public void _aload_0(int pc) {
    dumpPcNumber(pc);
    buffer.append(OpcodeStringValues
      .BYTECODE_NAMES[IOpcodeMnemonics.ALOAD_0]);
    writeNewLine();
}
```

The call to `dumpPcNumber` can obviously be realised as an aspect. However, in this case aspect-oriented programming could actually generate all 205 methods (including the comment) since the methods differ only in the name of the bytecode instruction.

4.1 Basic Approach for Data Collection

We implemented the ideas presented in Chapter 3 in a prototype called HAM (**H**istory-Based **A**spect **M**ining). It is powerful: it can identify potential aspects in large-scale software program. HAM analyses a software system's development history, obtaining sets of function calls that are likely to be cross-cutting. There is further potential in it: by informing the programmer unobtrusively when she is about to add more of that kind of functionality, the programmer can go on as planned, ignoring the 'warning', or think about introducing an abstraction of the functionality to encapsulate it properly.

The implementation of the prototype is based on a preprocessing that has been used by Zimmermann et al. for various other problems such as detecting common error patterns [56, 34]. Zimmermann et al.'s technique collects and preprocesses CVS data for fine-grained analysis of changes to a software system. This is why we chose their approach as the basis for HAM; we need to be able to analyse CVS data on the function-level. The preprocessing is useful since accessing a CVS archive for each analysis would be very time-consuming. Also, one does not always need all the information that could be obtained, such as type of change, author, date, time, and more. Other information that is useful for analyses, such as changes on a function-level, cannot be obtained directly from the CVS.

Preprocessing consists of several steps: data extraction, restoring transactions, and mapping changes to entities. A distinguishing attribute from usual approaches is that the whole preprocessing can be done incrementally. The data extracted in any earlier analysis run doesn't change. In other words, once more CVS data is available, only the new revisions have to be processed and added to the collected data rather than re-processing all of the CVS data.

The data-extraction phase stores all data in corresponding tables of a database: files, directories, revisions, transactions, tags, branches, author and log messages. Everything is extracted so that the data can be used for any kind of analysis, even though certain information might not be of interest for any current analysis. This leaves the filtering of unnecessary information up to the individual analysis using the preprocessed data.

The next phase deals with restoring transactions since CVS does not provide information about which files have been changed together in one commit to the version archive. However, this can be important information for certain analyses, and will be of importance for our aspect mining approach. Zimmermann et al. solve that problem by using a sliding time window over all changes that have been committed by the same author and with the same log message (if any). The time window is necessary to account for the fact that commits to version archives can take from a few seconds to several minutes, depending on the size of the files committed or the speed of the network connection.

As mentioned before, we will need data that enables us to do fine-grained analysis on a function-level which CVS does not provide. However, this information can be and is computed by comparing each revision's entities (on the function- rather than the file-level) with its predecessors. This way we can determine which functions have been modified, added, or deleted.

Our prototype uses this preprocessing of CVS data. Since we have every bit of information we need, starting from fine-grained changes on function-level to information about who the committing author is, we can mine the database following the techniques outlined in Chapter 3. An evaluation of our approach to analysis shows its effectiveness, which is presented in the following.

4.2 Evaluation Setup

Our mining approach can be applied to any version control system; however, we based our implementation on CVS, since most open-source projects use it. One of the major drawbacks of CVS is that commits are split into individual check-ins and have to be reconstructed. For this we use a *sliding time window* approach [56] with a 200-second window. A reconstructed commit consists of a set of revisions R where each revision $r \in R$ is the result of a single check-in.

Additionally, we need to compute method calls that have been inserted within a commit operation R. For this, we build abstract syntax trees (ASTs) for every revision $r_i \in R$ and its predecessor $r_{i-1} \in R$ and compute the set of all calls C_i in r_i and C_{i-1} for r_i's predecessor r_{i-1} by traversing the ASTs. Then $C_{r_i} = C_i \setminus C_{i-1}$ is the set of inserted calls within r_i; the union of all C_{r_i} for $r_i \in R$ forms a *transaction* $T = \bigcup_{r_i \in R} C_{r_i}$ which serves as input for our aspect mining and is stored in a database.

Since we analyse only differences between single revisions, there exists a drawback: we cannot resolve types because only one file is investigated at a time; in particular, we miss the signature of called methods. Of course, analysis would benefit from resolving types of identifiers. Unfortunately this requires usually analysing a file in the context of a complete program. As we analyse differences between single revisions of one file at a time, this context is not available and we cannot resolve types. ITo limit noise that is caused by this through name collision, we use the number of arguments in addition to method names to identify methods calls. This heuristic is frequently used when analysing single files [34, 53], and we believe it is suitable for our purposes since we are analysing one transaction at a time. We would get full method signatures when building *snapshots* of a system. However, as Williams and Hollingsworth [51] point out, such interactions with the build environment (compilers, make files) are extremely difficult to handle, require manual interaction, and result in high computational costs. In contrast, our preprocessing is cheap, as well as platform- and compiler-independent.

Renaming of a method is represented as deleting and introducing several method calls. Thus, we may incidentally consider renamed calls as aspect candidates. Recognising such changes is known as *origin analysis* [24] and will be implemented in a future version of HAM. It will eliminate some false positives and improve precision.

For a more thorough evaluation we chose three Java open-source projects and mined them for cross-cutting concerns. Refer to Table 4.2 for some statistics.

JHotDraw 6.0b1 is a GUI framework to build graphical drawing editors.[3] We chose it for its frequent use as an aspect mining benchmark.

Columba 1.0 is an email client that comes with wizards and internationalisation support.[4] We chose it because of its well-documented project history.

[3]`http://www.jhotdraw.org/`

[4]`http://www.columbamail.org/drupal/`

Eclipse 3.2M3 is an integrated development environment that is based on a plug-in architecture.[5] We chose it because it is a huge project with many developers and a large history.

For each project, we collected the CVS data as described in Section 4.1, mined for simple aspect candidates as defined in Section 3.2, reinforced them using the localities established in Section 3.4, and also built complex aspect candidates as introduced in Section 3.2. We investigated the following questions:

1. *Simple Aspect Candidates.* How precise is our mining approach? That is, how many simple aspect candidates are real cross-cutting concerns?

2. *Reinforcement.* This leads to larger aspect candidates, but does it actually improve precision (by ranking true simple aspect candidates high)?

3. *Ranking.* Can we rank aspect candidates such that more cross-cutting concerns are ranked high? Which technique finds cross-cutting concerns better?

4. *Complex Aspect Candidates.* How many complex aspect candidates can we find by the combination of simple ones?

To measure *precision*, we computed for each project, ranking, and reinforcement algorithm the top 50 simple aspect candidates. Since some aspect candidates might rank within the top 50 in more than one ranking for the same project, there are possible duplicates in the rankings. To avoid multiple evaluation effort for those, we combined these rankings into one set per project. For Columba we got 134, for Eclipse 159, and for JHotDraw 102 unique simple aspect candidates. Next, we sorted these sets alphabetically by the name of the called method in order to prevent bias in the subsequent evaluation, e.g., by knowing which candidates came from which ranking. We used this order to classify simple aspect candidates manually into *true* and *false* cross-cutting concerns. The *precision* is then defined as the ratio of the number of true cross-cutting concerns to the number of aspect candidates that were uncovered by HAM. Precision is the accuracy of our technique's results and in general a common measure for search performance.

We considered an aspect candidate (M, L) as a true cross-cutting concern if it referred to the same functionality and the methods M were called in a similar way, i.e., at the same position within a method and with the same parameters. An additional requirement for a true cross-cutting concern was that it can be implemented using AspectJ. However, we did not take into account whether aspect-orientation is the best way to realise the given functionality. In cases of doubt, we classified a candidate as a false cross-cutting concern.

Of course, it would also be interesting to measure how many of all existing aspect candidates have been detected. *Recall* is the ratio of the number of correct target predictions (i.e., correctly identified aspect candidates) and the number of all target examples (i.e., all candidates). Thus, it measures how well a search algorithm (such as our approach) finds what is is supposed to find. However, determining recall values

[5]`http://www.eclipse.org/`

Table 4.1: Precision of HAM (in %) for simple aspect candidates

	Columba	**Eclipse**	**JHotDraw**
Size	52	52	36
Fragmentation	46	54	30
Compactness	42	52	28

Table 4.2: Evaluation subjects

	Columba	**Eclipse**	**JHotDraw**
Presence			
Lines of code	103 094	1 675 025	57 360
Java files	1 633	12 935	974
Java methods	4 191	74 612	2 043
History			
Developer	19	137	9
Transactions	4 105	97 900	269
– that changed Java files	3 186	77 250	241
– that added method calls	1 820	43 270	132
Method calls added	24 623	430 848	7 517
First transaction	04-08-2001	05-02-2001	10-12-2000
Last transaction	11-02-2005	11-23-2005	04-25-2005

requires the knowledge of all aspect candidates which is impossible for real-world software systems. The projects that we analyse are too large to calculate recall, or more precisely, the number of target examples, by hand, and it is not possible to even speculate about recall as the presented analysis approach is completely new. We therefore cannot report recall numbers.

4.3 Simple Aspect Candidates

To evaluate our notion of simple aspect candidates we checked via manual inspection whether the top-50 candidates per ranking and project were cross-cutting or not. The precision as the ratio of true cross-cutting functionality and all (50) aspect candidates is listed in Table 4.1 on page 39 for each project (columns) and each ranking (rows).

We observe that precision increases with subject size: it is highest for Eclipse and lowest for JHotDraw, the smallest subject. The ranking has a minor impact and no ranking is generally superior; the deviation among the precision values is at most 10 percentage points. Nevertheless, the ranking by size, which simply ranks by the number of locations where a method was added, seems to work well across all projects. It reaches a precision between 36 and 52 percent. Roughly speaking, every second (for JHotDraw every third) mined aspect candidate is a real cross-cutting concern.

Unlike ranking by size, ranking by fragmentation and by compactness take trans-

Table 4.3: The effect of reinforcement on the precision of HAM (in % points)

	Columba	**Eclipse**	**JHotDraw**
Temporal locality			
Size	+ 2	− 4	± 0
Compactness	+ 2	− 2	+ 4
Possessional locality			
Size	− 8	−20	+ 2
Compactness	+12	+ 8	+ 2
All localities			
Size	− 8	−20	+ 2
Compactness	+10	+ 6	+ 2

actions or the overall number of modified locations into account. We believe that the poor performance of these rankings for our smaller subjects JHotDraw and Columba is caused by the much smaller number (hundreds/few thousands versus tens of thousands) of transactions and added method calls available for mining (see Table 4.2). In other words, we expect these rankings to benefit from long project histories. These generally correspond to many transactions, as present in Eclipse.

To summarise, as answer to our first evaluation question regarding simple aspect candidates, we can say that *we were able to identify cross-cutting concerns with a precision of between 36% and 54%; the precision increases with project size and history, i.e., large projects with long history boost precision.* The cost of false positives is the time a developer invests in examining them.

4.4 Reinforcement

After mining simple aspect candidates we evaluated the effect of reinforcement on them. Reinforcement takes a simple aspect candidate (M, L) from a single transaction and looks at locally related transactions in order to arrive at a candidate (M, L') with an enlarged set $L' \supset L$ of locations. For the evaluation we reinforced the simple aspects of our subjects using temporal, possessional, and also using all localities applied at once. As before, we checked the top-50 aspect candidates and computed the precision.

Table 4.3 lists the *change in precision* for each subject (columns), each locality (rows), and each ranking by size or compactness (sub-rows). Changes are relative to the precision before reinforcement (Table 4.1). Hence, these changes express the effect of reinforcement on the precision of our mining.[6]

Temporal locality produces slight improvements but seems to be unsatisfying with large projects. We presume that this is because we chose the same fixed time window of 2 days for all three subjects; we plan to investigate whether a window size proportional to a project's size would yield better results. The Eclipse project has far more

[6]Note that for reinforcement we did not rank by fragmentation. This ranking punishes reinforced aspect candidates that are spread across many transactions.

developers as well as CVS transactions per day than JHotDraw and Columba. Thus, we have too much noise that diminishes the positive impact of temporal locality for Eclipse.

Possessional locality shows the most significant improvement. Albeit ranking by size decreases precision up to 20 percentage points, possessional locality in combination with ranking by compactness improves precision up to 12 percentage points for all three subjects. In large projects, `get` and `set` methods are inserted in many locations and thus alleviate the positive effects of possessional locality for Eclipse when aspect candidates are ranked by size.

All localities considers the application of both localities. The effect on the precision is the same as with reinforcement based on possessional locality only: ranking by size annihilates the positive impact, ranking by compactness facilitates it. Thus, possessional locality is dominant and affects precision prominently.

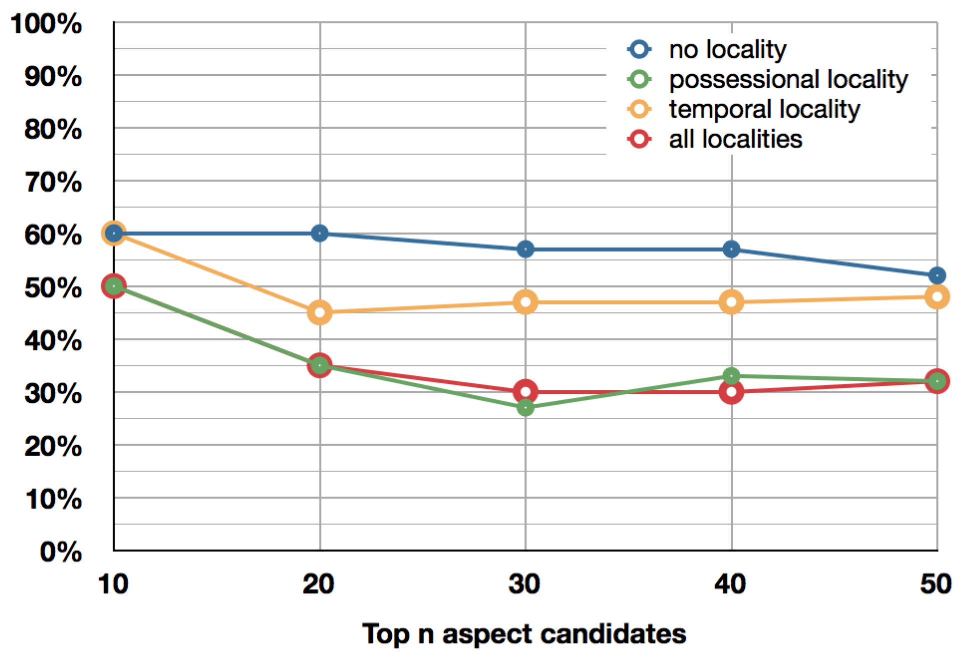
The good results for possessional locality suggest that aspects belong to a developer, and are mostly not distributed over many transactions. This is backed up by the notably improved precision of our approach after reinforcement based on possessional locality combined with ranking by compactness. Besides, all our results, without and with reinforcements, suggest that small projects have small histories and thus we achieve a considerably lower precision. In addition, precision can only be improved marginally with reinforcements. This is of some consequence as reinforcements leverage a large number of transactions and developers. In contrast, projects such as Eclipse with a long and large history do extremely well.

To summarise, as answer to our second question posed in this evaluation regarding the impact of reinforcement, we can say that *possessional locality improves the precision for ranking by compactness; this indicates that cross-cutting concerns are owned by developers.*

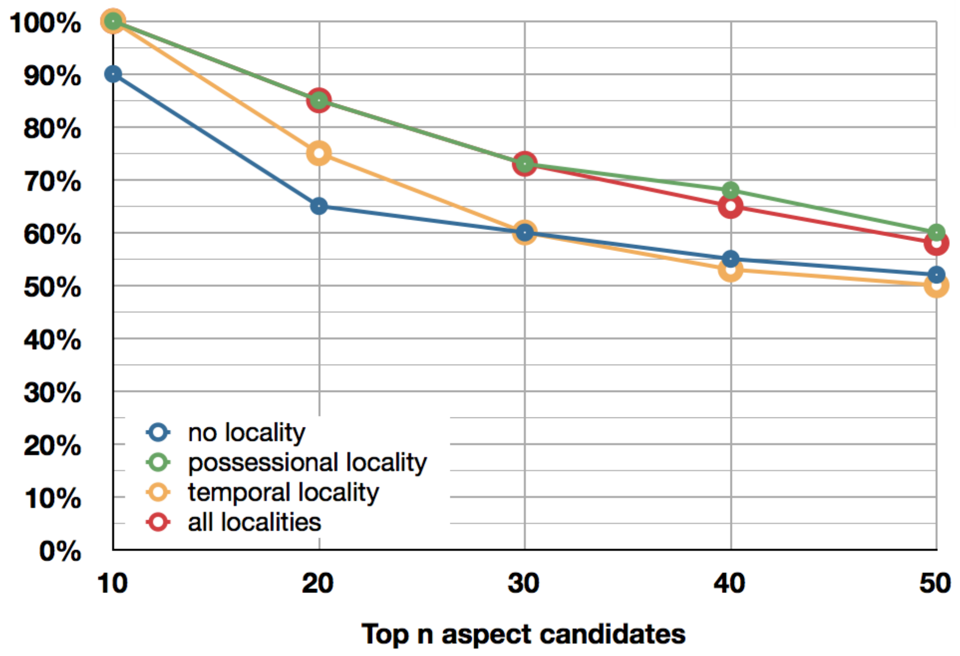
4.5 Precision Revisited

So far we have evaluated our mining by computing the precision of the top-50 aspect candidates in a ranking. However, it is unlikely that a developer is really interested in 50 aspect candidates. Based on experience we know that she will instead probably look only at ten or twenty candidates at most. We have therefore broken down the precision for the top ten, twenty, and so on candidates for each project. The results for all three subjects are similar. For the detailed discussion here, we have chosen Eclipse for two reasons—it is an industrial-sized project and the results are most meaningful; they are plotted in Figure 4.1. For the sake of completeness, we have included the results for Columba and JHotDraw in Figures 4.2 (page 43), and 4.3 (page 43).

The graph in Figure 4.1(a) shows the precision when *ranked by size* before and after applying different reinforcements. The precision stays mostly flat when moving from the top-50 to the top-10 candidates. However, the overall precision remains between 30 and 60 percent. Reinforcement seems to make matters only worse, as ranking by size before reinforcement performs best. We can observe roughly the same for the small projects Columba and JHotDraw (see Figures 4.2(a) and 4.3(a)).

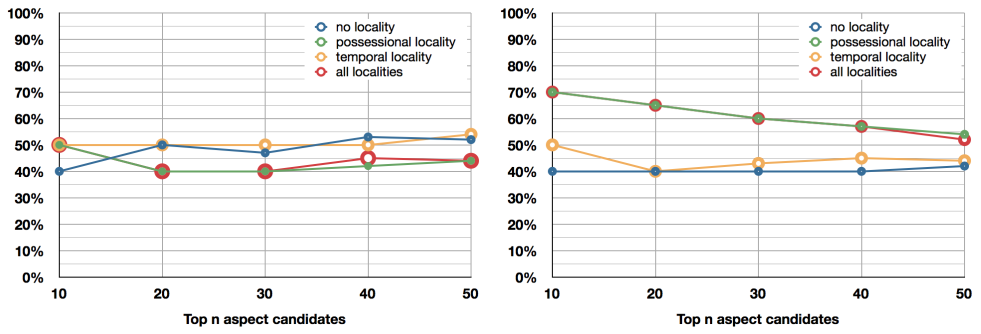


(a) Dependent on length of ranking by size



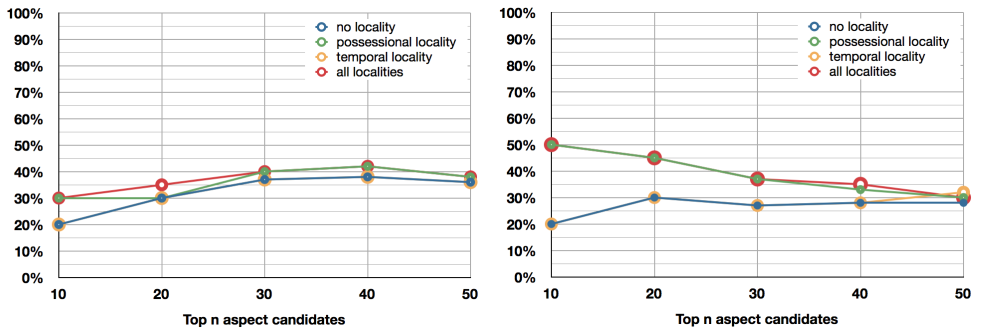
(b) Dependent on length of ranking by compactness

Figure 4.1: Precision of HAM for Eclipse



(a) Dependent on length of ranking by size (b) Dependent on length of ranking by compactness

Figure 4.2: Precision of HAM for Columba



(a) Dependent on length of ranking by size (b) Dependent on length of ranking by compactness

Figure 4.3: Precision of HAM for JHotDraw

In contrast, Figure 4.1(b) shows a dramatically different picture for the precision when *ranked by compactness*. The precision is highest for the top-10 candidates and decreases when additional candidates are taken into account; it is lowest for the top-50 candidates. However, the first ten candidates have a precision of at least 90%. This means, nine out of ten are true cross-cutting concerns. Thus, ranking by compactness is very valuable for developers. Again, a similar improvement can be observed for Columba and JHotDraw in Figures 4.2(b) and 4.3(b): precision for the top ten candidates increases from 30% (JHotDraw) and 50% (Columba) to 50% and 70% respectively.

In summary, size is not the most prominent attribute of cross-cutting concerns, but compactness is. This is also supported by the obversation that temporal and possessional locality enhance ranking by compactness for Eclipse. For Columba and JHotDraw, temporal locality has not as much impact as possessional locality. This is probably due to the fact that the development history for both projects is not as long and extensive as it is the case for Eclipse. It confirms our earlier findings (see Section 4.4) which showed that cross-cutting concerns are owned by developers and thus possessional locality improves precision.

So, to answer our third question in this evaluation, whether ranking can give us true aspect candidates first, we can summarise our findings by saying that *ranking by compactness pushes true cross-cutting concerns to the top such that we reach a precision of 90% for the top-10 candidates in the industrial-sized system Eclipse.*

4.6 Complex Aspect Candidates

For our evaluation subjects, we combined simple aspect candidates into a complex candidate if they cross-cut exactly the same locations. This condition was very selective: for Columba we got 21, for Eclipse 178, and for JHotDraw 11 complex aspect candidates. Note that all candidates cross-cut at least 8 locations. Below, we discuss the results from Eclipse in more detail.

Table 4.4 shows the top 20 complex aspect candidates ranked by size for the Eclipse project. Each row represents one complex aspect candidate (M, L). The second column contains the methods M called by an aspect candidate, where the number in brackets denotes the number of arguments for each method. The third column gives the number $|M|$ of methods and the fourth column shows the number $|L|$ of method locations where calls to M were inserted. In the first column we provide the result of our manual inspection of this aspect candidate: ✓ for an actual cross-cutting concern and ✗ for a false positive.

HAM indeed finds cross-cutting concerns consisting of several method calls. In addition, they are ranked on top of the list. However, the performance of our approach decreases when it comes to lower-ranked aspect candidates. We believe that one reason for poor performance are `get` and `set` methods that are inserted in many locations at the same time and thus out-rank actual cross-cutting concerns in the number of occurrences. Although these getters and setters are not cross-cutting, they still describe perfect usage patterns.

Furthermore, we find only few complex cross-cutting concerns. This is mainly a consequence of the condition that the locations sets have to be the same ($L_1 = L_2$). We could relax this criterion to the requirement that one location set has to be a subset of the other ($L_1 \subseteq L_2$), however, this adds exponential complexity to the determination of aspect candidates. We will improve on this in our future work. For now, let us look at three cross-cutting concerns in Eclipse.

Locking Mechanism. This cross-cutting concern was already mentioned in Chapter 1. Calls to both methods `lock` and `unlock` were inserted in 1 284 method locations. Here is such a location:

```
public static final native void _XFree(int address);
public static final void XFree(int /*long*/ address) {
    lock.lock();
    try {
        _XFree(address);
    } finally {
        lock.unlock();
    }
}
```

The other 1 283 method locations look similar. First `lock` is called, then a corresponding native method, and finally `unlock`. It is a typical example of a cross-cutting concern which can be easily realised using AOP. Note that this `lock/unlock` concern cross-cuts different platforms. It appears in both the GTK and Motif version of Eclipse. Typically such cross-platform concerns are recognised incompletely by static and dynamic aspect mining approaches unless the platforms are analysed separately and results combined.

Abstract Syntax Trees. Eclipse represents nodes of abstract syntax trees (ASTs) by the abstract class `ASTNode` and several subclasses. These subclasses fall into the following simplified *categories*: expressions (`Expression`), statements (`Statement`), and types (`Type`). Additionally, each subclass of `ASTNode` has *properties* that cross-cut the class hierarchy. An example for a property is the *name* of a node: There are named (`QualifiedType`) and unnamed types (`PrimitiveType`), as well as named expressions

Table 4.4: Complex aspect candidates (M,L) found for Eclipse

	M	$\lvert M\rvert$	$\lvert L\rvert$
✓	{`lock(0)`, `unlock(0)`}	2	1284
✓	{`postReplaceChild(3)`, `preReplaceChild(3)`}	2	104
✓	{`postLazyInit(2)`, `preLazyInit(0)`}	2	78
✗	{`blockSignal(2)`, `unblockSignal(2)`}	2	63
✓	{`getLength(0)`, `getStartPosition(0)`}	2	62
✓	{`hasChildrenChanges(1)`, `visitChildrenNeeded(1)`}	2	62
✗	{`modificationCount(0)`, `setModificationCount(1)`}	2	60
✗	{`noMoreAvailableSpaceInConstantPool(1)`, `referenceType(0)`}	2	57
✗	{`g_signal_handlers_block_matched(7)`, `g_signal_handlers_unblock_matched(7)`}	2	54
✗	{`getLocalVariableName(1)`, `getLocalVariableName(2)`}	2	51
✗	{`isExisting(1)`, `preserve(1)`}	2	48
✗	{`isDisposed(0)`, `isTrue(1)`}	2	37
✗	{`gtk_signal_handler_block_by_data(2)`, `gtk_signal_handler_unblock_by_data(2)`}	2	34
✗	{`error(1)`, `isDisposed(0)`}	2	31
✗	{`getWarnings(0)`, `setWarnings(1)`}	2	31
✗	{`getCodeGenerationSettings(1)`, `getJavaProject(0)`}	2	31
✗	{`SimpleName(1)`, `internalSetIdentifier(1)`}	2	29
✗	{`iterator(0)`, `next(0)`}	2	27
✓	{`postValueChange(1)`, `preValueChange(1)`}	2	26
✗	{`SimpleName(1)`, `internalSetIdentifier(1)`}	2	25

(`FieldAccess`). Additional properties of a node include the *type*, *expression*, *operator*, or *body*.

This is a typical example of a *role super-imposition* concern [36]. As a result, every named subclass of `ASTNode` implements method `setName` which results in duplicated code. With AOP the concern could be realised via the method-introduction mechanism.

```
public void setName(SimpleName name) {
    if (name == null) {
        throw new IllegalArgumentException();
    }
    ASTNode oldChild = this.methodName;
    preReplaceChild(oldChild, name, NAME_PROPERTY);
    this.methodName = name;
    postReplaceChild(oldChild, name, NAME_PROPERTY);
}
```

Our mining approach revealed this cross-cutting concern with several aspect candidates. The methods `preReplaceChild` and `postReplaceChild` are called in the aforementioned `setName` method; the methods `preLazyInit` and `postLazyInit` guarantee the safe initialisation of properties; and the methods `preValueChange` and `postValue-Change` are called when a new operator is set for a node.

Cloning. Another cross-cutting concern was surprising because it involved two getter methods `getStartPosition` and `getLength`. These are always called in `clone0` of subclasses of `ASTNode` and were also identified by our approach.

```
ASTNode clone0(AST target) {
    BooleanLiteral result = new BooleanLiteral(target);
    result.setSourceRange(this.getStartPosition(),
                          this.getLength());
    result.setBooleanValue(booleanValue());
    return result;
}
```

To summarise our findings for the last question in our evaluation regarding the identification of complex aspect candidates, we can say that *we do indeed find complex cross-cutting concerns. They are rare but they represent serious maintenance challenges without AOP. Once again, the complex aspect candidates are ranked high.*

4.7 Conclusions

Our evaluation confirms the hypothesis underlying our approach: cross-cutting concerns do not necessarily exist from the beginning but emerge over time. They might even 'die out' over time when they are only temporarily introduced to deal with problems arising in the development process. The hypothesis is further supported by the observation that the more history we have the more precise we are. By introducing and formalising the dimension of time for aspect mining, we developed an approach that has several advantages. Mining version archives is an effective way to mine for cross-cutting concerns.

We can successfully identify cross-cutting concerns although better precision is achieved the larger a project's size is, the more developers it has, and the more revisions it features. Simply put, the more history we can analyse, the more precise we can be (e.g., we reached a precision of 90% for the top-10 candidates in Eclipse). If less project data is available, for example for small software systems, we only reach a precision comparable to previous approaches (about 60%). However, the fact that we scale to industrial-sizes projects with a high precision is a major advantage over existing aspect mining approaches.

Recalling the cross-cutting concern we discovered and mentioned in Chapter 1, the implementation of a new `lock/unlock` mechanism in Eclipse, we also see that we are able to discover cross-cutting concerns across platform-specific code. Previous approaches, whether static or dynamic, identify such cross-cutting concerns only by mining the code base repeatedly.

Applying different concepts of locality as well as different ranking techniques showed that some of them, even though intuitive, produce noise which can be counterproductive, but overall they improve the precision of our approach. Most notably, ranking by compactness showed to be the best choice with large projects such as Eclipse. Our evaluation showed that its precision is aided by choosing possessional locality. This fact gives additional insight into software development projects: it indicates that cross-cutting concerns are owned by developers, and one concern is usually dealt with by one and the same developer, even if later changes have to be applied.

Overall, our evaluation has shown that mining cross-cutting concerns from version archives is an effective and precise new approach with its scalability to industrial-sized projects being a great asset.

"Analysis is the art of taming infinity."

Neil Falkner

5 Formal Concept Analysis

As we have discussed before, software systems may contain functionality that does not align well with its overall architecture but cross-cuts instead. In the previous chapters we have proven our hypothesis that these do not necessarily exist from the beginning but emerge over time. By analysing where developers add code to a program, our history-based mining identifies cross-cutting concerns in a two-step process. First, we mine CVS archives for sets of methods where a call to a specific single method was added. In a second step, such simple cross-cutting concerns are combined to complex cross-cutting concerns. To compute these efficiently, we now apply formal concept analysis—an algebraic theory.

5.1 Basics of Formal Concept Analysis

Formal concept analysis (FCA) is used for exploratory data analysis problems and requires no a priori knowledge. Usually, the data consists of relationships $\mathcal{R}$ between objects $\mathcal{L}$ and attributes $\mathcal{M}$. Such a data set contains conceptual clusters (called *(formal) concepts* in the context of FCA) where a cluster is a set of objects that all share the same attributes. The concepts (or clusters) that exist in the data set, form a partially ordered hierarchy, which is a so-called *concept lattice*: $(L_1, M_1) \leq (L_2, M_2) \iff L_1 \subseteq L_2$. This means that any two concepts have a unique largest common sub-concept, and any intersection of two concepts is a concept in the lattice, too.

Ganter and Wille's [22] algorithm for concept analysis solves the problem of computing efficiently all concepts in a data set. What it does not provide explicitly is the resulting concept lattice. Lindig [33] developed an efficient and all-purpose concept analysis algorithm. It not only computes all concepts as well as the concept lattice, but does so in a breadth-first fashion which allows one to explore only the top-most concepts, or to stop if a given minimum support is not reached any longer. We will see the use of the latter feature in Section 5.2 where we will introduce a minimal size that must be exceeded for a concept to be of interest for our aspect mining problem. The basic principle of Lindig's algorithm is described in the following.

The starting point is the usually empty top concept (or cluster) for a relationship $\mathcal{R} \subseteq \mathcal{L} \times \mathcal{M}$, often denoted with $\top$. The reason for it to usually be empty is that it is highly unlikely that all objects in the data set share at all share one and the same attribute(s). Similarly, the bottom node is empty and denoted with $\perp$ since for it to not be empty the data set would need to contain at least one object which shares all attributes at once. For any given concept (L, M) the algorithm computes a sub-concept $(L_m, M_m) = ((M \cup \{m\})', (M \cup \{m\})'')$ for each attribute $m \in \mathcal{M} \setminus M$ that is not already in (L, M). This list of sub-concepts contains all lower neighbours of (L, M) in the lattice but can contain additional concepts. The following criterion only holds for lower neighbours and is used to identify them: (L_m, M_m) is a lower neighbour iff $(M \cup \{x\})'' = (M \cup \{m\})''$ holds $\forall x \in M_m \setminus M$. The '-operator (also *prime operator*) can be understood as derivative function. The derivative of a set of objects $L_i \subseteq L$ is the set $L'_i \subseteq M$ of all attributes that hold for all objects in L_i, and the derivative of a set of attributes $M_i \subseteq M$ is the set $M'_i \subseteq L$ of all objects that have all attributes in M_i.

Details about the algorithm, including proofs, can be found in [33]. In the following we will illustrate how formal concept analysis can be used to compute single and complex aspect candidates efficiently.

5.2 Mining Cross-Cutting Concerns

To recap, we model the history of a program as a sequence of transactions. A *transaction* collects all code changes between two versions, called *snapshots*, made by a programmer to complete a single development task. Within each transaction we are searching for *added method calls* which may identify an aspect. We consider calls to a small set of (related) methods that are added in many (unrelated) locations a cross-cutting concern or *aspect candidate*.

We refer to a method where calls to another method are added as *location*, and to the method being called simply as *method*. An aspect candidate is thus characterised by two sets: a set of locations and a set of methods. This definition represents a trade-off: albeit it is not fully general, it still captures many interesting cross-cutting concerns and enables us to identify them efficiently. This trade-off is particularly worthwhile when it comes to finding complex aspect candidates and possible mutations of them that we might miss if we derive complex candidates only by following the restrictive combination of simple aspect candidates as defined in Chapter 3.

Aspects are maximal Blocks. We can think of a transaction as a cross table with locations as rows and methods as columns (Figure 5.1, left). The intersection of location l and method m is marked with a cross when the transaction inserts a call to m in location l. In this representation, each column is a simple aspect candidate; however, to cut out noise, we only consider columns with at least 7 crosses. Formally,

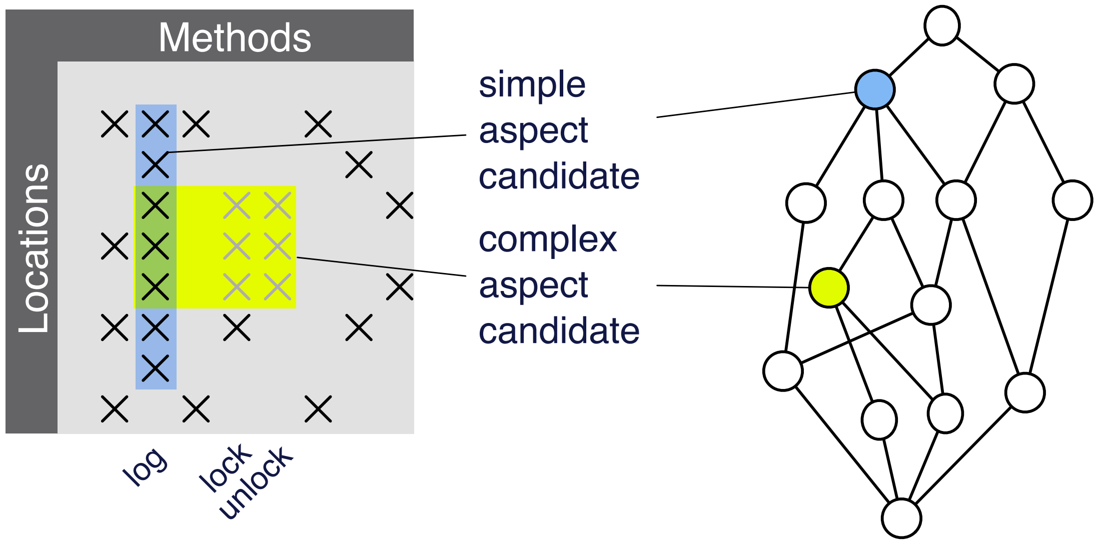


Figure 5.1: Maximal blocks represent aspect candidates in a transaction (left). Here, 14 candidates form a lattice of super- and sub-aspects (right). A sub-aspect (yellow) crosscuts fewer locations but calls more methods than a super-aspect (blue).

a simple candidate is a pair (L, M) of locations L and methods M with $|M| = 1$ and $|L| \geq 7$.[7]

Given a specific simple aspect candidate (L, M), we can arrange the table such that all rows from L are adjacent to each other. Now a simple aspect candidate manifests itself as a *maximal block* in the table of width $|M| = 1$ and height $|L|$. In Figure 5.1 on page 51 such a block is marked by the blue-shaded rectangle of size 1×7. A *complex aspect candidate* (L, M) is a maximal block with $|M| > 1$: at each location $l \in L$ all methods $m \in M$ are called. An example is the second, yellow-shaded rectangle of size 3×3 in Figure 5.1. However, to obtain such a block for a complex aspect candidate in general, we have to re-order not just rows but also columns. It is therefore not obvious how to compute all blocks present in a transaction.

Identifying maximal blocks in a cross table (or transaction) $T \subseteq \mathcal{L} \times \mathcal{M}$ is provided by the algebraic theory of formal concepts as introduced in Section 5.1. Note that in our case $\mathcal{L}$ is identical to $\mathcal{M}$ since a method's name identifies its body, and thus the location where a method call has been added. A maximal block is a pair (L, M) where the following holds:

$$\begin{array}{rcl} L & = & \{l \in \mathcal{L} \mid (m, l) \text{ for all } m \in M\} \\ M & = & \{m \in \mathcal{M} \mid (m, l) \text{ for all } l \in L\} \end{array}$$

Each block (L, M) is maximal in the following sense: we cannot add another method

[7]Note that in this chapter we use the notation (L, M) for 'method m was added in location l' whereas in Chapter 3 we used (M, L) to express the same thing. This change of order is required to accommodate aspects to the structure of the formal concept analysis, but does not change the meaning of such pairs representing aspect candidates.

m to M without shrinking L to ensure that *all* locations in L call m. Likewise, we cannot add another location l to L without shrinking M. The definition allows for blocks of any size though for our purposes we only consider blocks representing simple aspect candidates with $|L| \geq 7$. To identify the most interesting ones, we additionally take the *area* $|L| \times |M|$ of a block as a measure.

In the worst case, a transaction may contain exponentially many blocks. This makes concept analysis potentially expensive–even in the presence of efficient algorithms [33]. This is not a concern here since we compute the blocks for each transaction individually, thus apply FCA to many but relatively small sets of data. Computing all blocks for the 43 270 transactions of Eclipse took about 43 seconds, that is, about one millisecond per transaction.

The aspect candidates of a transaction form a lattice given the following partial order: $(L, M) \leq (L', M')$ iff $L \subseteq L'$. A sub-aspect cross-cuts fewer locations than its super-aspect but calls more methods (cf. Figure 5.1, graph on the right). In our experience, aspects in one transaction are rarely in a super/sub order, but are typically unordered.

5.3 Example Results from Eclipse

Figure 5.2 on page 53 shows the lattice of all aspect candidates from an EclipseCVS commit transaction on 2004-03-01. In the lattice two aspects are connected if they are in a direct super/sub-concept relation. Nodes are given the shape of the corresponding block which gives prominence to large aspect candidates: For example, candidate 6 contains 14 location where calls to `unsupportedIn2()` were added. This method throws an exception if the operation called is not supported at API level 2.0.

```
public void setName(SimpleName name) {
        if (name == null) {
                throw new IllegalArgumentException();
        }
        ASTNode oldChild = this.methodName;
        preReplaceChild(oldChild, name, NAME_PROPERTY);
        this.methodName = name;
        postReplaceChild(oldChild, name, NAME_PROPERTY);
}
```

An even larger example for a cross-cutting concerns is the following: Eclipse represents nodes of abstract syntax trees by the abstract class `ASTNode` and several subclasses. These subclasses fall into the following simplified *categories*: expressions (subclass `Expression`), statements (subclass `Statement`), and types (subclass `Type`). Additionally, each subclass of `ASTNode` has *properties* that cross-cut the class hierarchy. An example for a property is the *name* of a node: there are named (`QualifiedType`) and unnamed types (`PrimitiveType`), as well as named expressions (`FieldAccess`). Additional properties include the *type*, *expression*, *operator*, or *body* that are associated with a node in an abstract syntax tree.

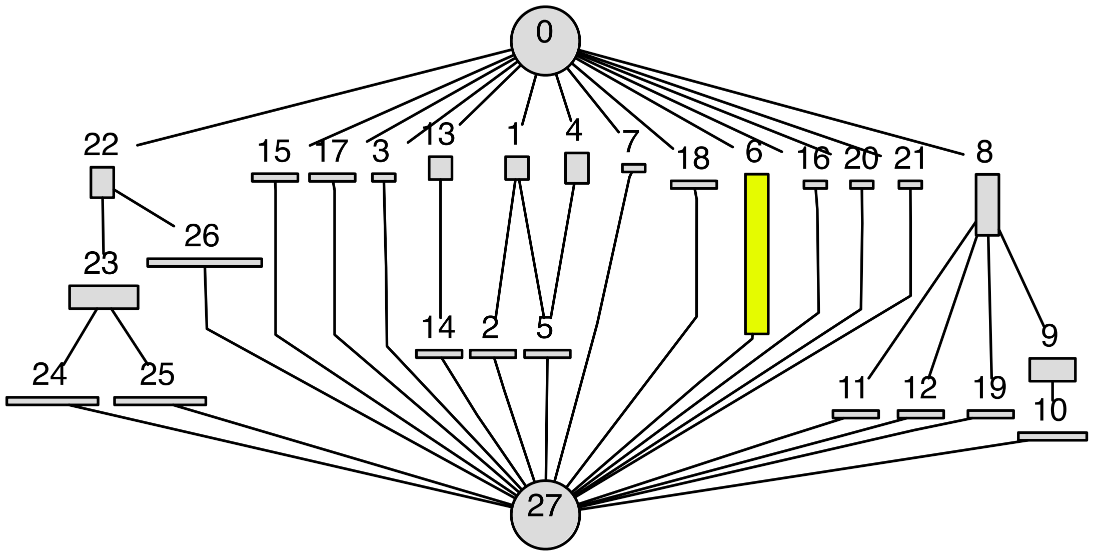


Figure 5.2: The lattice of aspect candidates from a commit to EclipseCVS on 2004-03-01 by developer `ptff`. Candidate 6 contains 14 additions of calls to `unsupportedIn2()`.

This is a typical example for a *role super-imposition* concern [36]. As a result of this cross-cut, every named subclass of `ASTNode` implements the method `setName` which results in duplicated code that is difficult to maintain. With aspect-oriented programming the concern could be realised with the method introduction mechanism.

Our mining approach revealed this cross-cutting concern with several aspect candidates. The lattice for the corresponding commit transaction is shown in Figure 5.3 on page 54.

The methods `preReplaceChild` and `postReplaceChild` are called in the aforementioned `setName` method and many other methods. Node 10 contains 104 locations where calls to both methods are added. The methods `preLazyInit` and `postLazyInit` guarantee the safe initialisation of properties and calls to them are added in 78 locations; node 11 is the corresponding node in the lattice in Figure 5.3. The methods `preValueChange` and `postValueChange` are called when a new operator is set for a node; calls to them have been added in 26 locations, represented by node 12 in the lattice.

Summary. As we mentioned before, concept analysis is potentially expensive since a cross table of size $n \times n$ (since methods are also the locations) may have up to 2^n blocks. However, this has not been a problem; the 43 270 transactions of the Eclipse CVS archive constitute 159 448 blocks. The average transaction adds 5.4 calls in 3.8 locations and has 3.7 blocks. The largest transaction had 1 235 blocks (and was the anecdote we identified with the original approach and mentioned in the Introduction). It took on average less than 1 second for Lindig's algorithm to compute all blocks for a transaction.

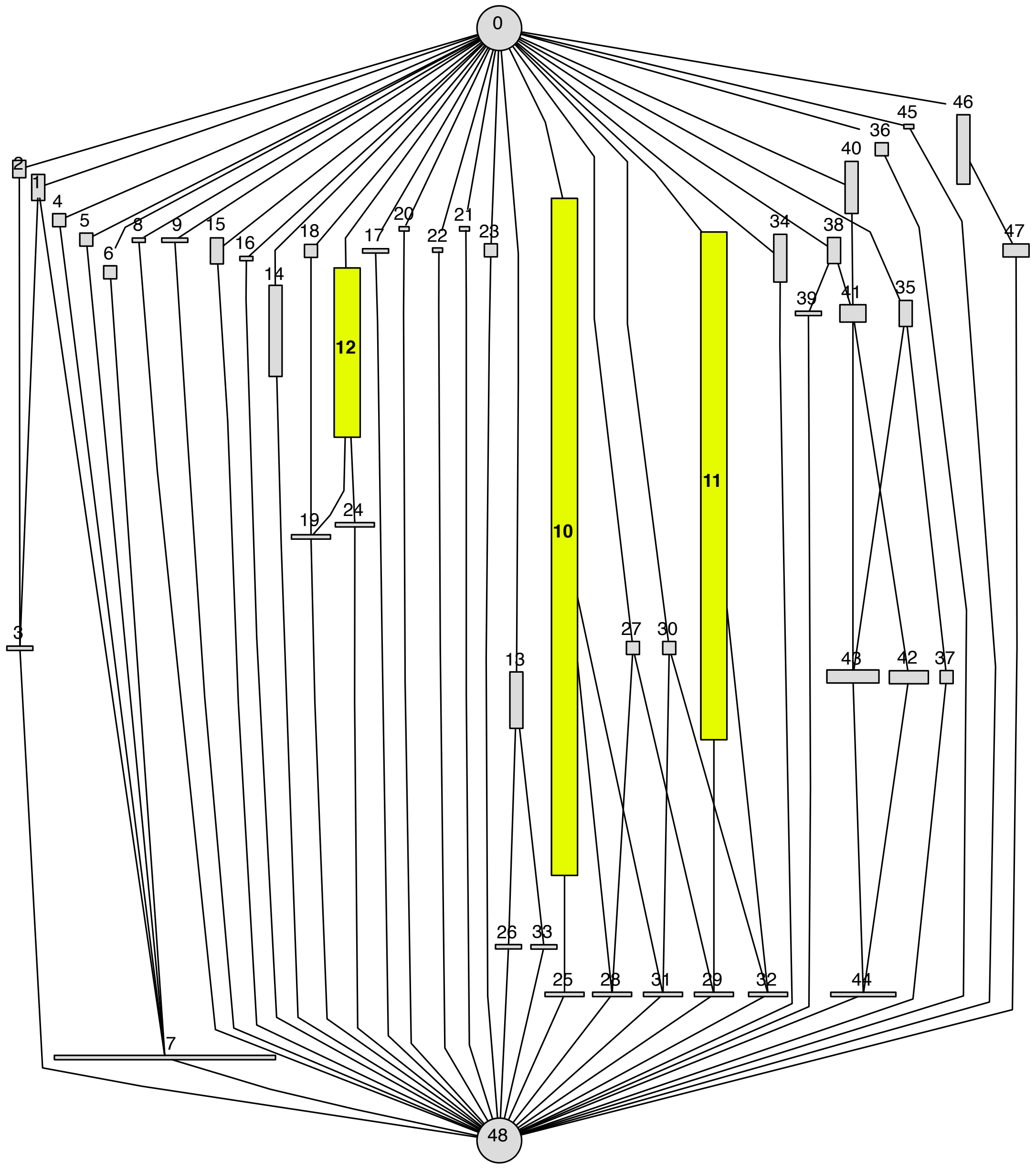


Figure 5.3: The lattice of aspect candidates from a commit to EclipseCVS on 2004-02-25 by developer `ptff`. Candidate 10, e.g., contains 104 additions of calls to `preReplace-Child(3), postReplaceChild(3)`.

Aspect Candidates in Eclipse 3.2M3				
methods	1	2	3	≥ 4
candidates	1878	363	88	24

Table 5.1: Aspect candidates mined from 43 270 CVS transactions for Eclipse 3.2M3. There are 88 candidates that added exactly three method calls.

We mined 2 353 aspect candidates, with the distribution shown in Table 5.1. We found 1 878 simple and 363+88+24 = 475 complex candidates. In [8] we had previously mined Eclipse for simple and complex aspect candidates, albeit with a less general approach. We found 31 unique complex candidates that cross-cut at least 20 locations (out of which we found 6 to be true aspects and additional 3 to be partial aspects). Using formal concept analysis we found 64 unique aspect candidates, including all 31 aspect candidates reported in [8]. This confirms that formal concept analysis provides the correct formal and algorithmic framework to mine for aspects. Furthermore, the mining can be done efficiently from large projects by analysing code additions over time.

"We can't solve problems by using the same kind of thinking we used when we created them."
Albert Einstein

6 Representing Concerns in Code and Inferring Structural Patterns

Nowadays, software systems are complex, and thus the features or concerns of which they are comprised are not necessarily always encapsulated in their own module, e.g., a class[8]. There are various reasons for this phenomenon, and not all of them relate to developers' ability to design software in a modular fashion. In particular, with increasingly complex systems, it is inherently harder to achieve a modular design: some concerns are difficult to encapsulate [29], while others are prone to scattering as the result of repeated changes [18].

If source code implementing a concern is scattered, a programmer who has to make modifications to this concern not only has to spend time and effort locating the concern's code; she just also understand it to be able to perform the necessary changes. Concern location is an important software engineering task, and many approaches have been introduced to assist this tricky activity. Examples include the approach of Eisenbarth et al., who propose to associate source code with features based on the analysis of execution traces [19], or SNIAFL, a technique that associates functions with feature descriptions based on information retrieval techniques [55]. Several aspect mining approaches fall into the category of concern location, too, such as our own approach presented in the previous chapters. Independently of the exact details of the concern location technique used, the result of the feature location task will be a list of program elements (e.g., methods and fields in an object-oriented language) that are associated with the concern in which the developer is interested.

Concern location is clearly a demanding task which can take up a substantial amount of a programmer's time, depending on the technique applied, the feature under investigation, and the developer's familiarity with the code base. Hence, it would be

[8]For present purposes, we use *concern* and *feature* interchangeably, even though strictly a feature is a particular form of a concern.

worth finding a way to preserve any results of feature location efforts as an artifact of the software project so that it can be re-used and consulted later when needed; it would be equally useful to the developer who worked on it originally and to others.

Unfortunately, a concern-code mapping, also simply referred to as a *mapping*, can become void as soon as changes are made to the program code of the software system and become permanent when submitted to the revision control system. And it is not only the big changes that can destroy one's previous efforts in reverse engineering. Even trivial changes such as simply renaming a method can potentially invalidate feature location results, since it is not necessarily straightforward to rediscover the original name. Since one of our research foci is on aspect mining, and thus on feature location, we are keen to identify ways that can help preserve knowledge acquired via these activities as lastingly and as accurately as possible.

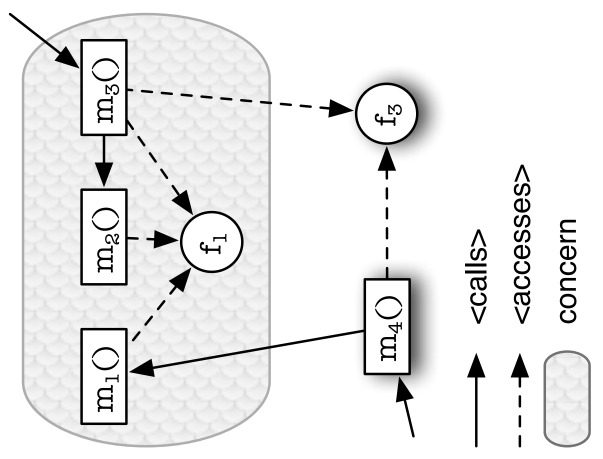
This as well as the following chapter is a rework of the paper on Inferring Structural Patterns for Concern Traceability in Evolving Software [13].

6.1 Basic Idea

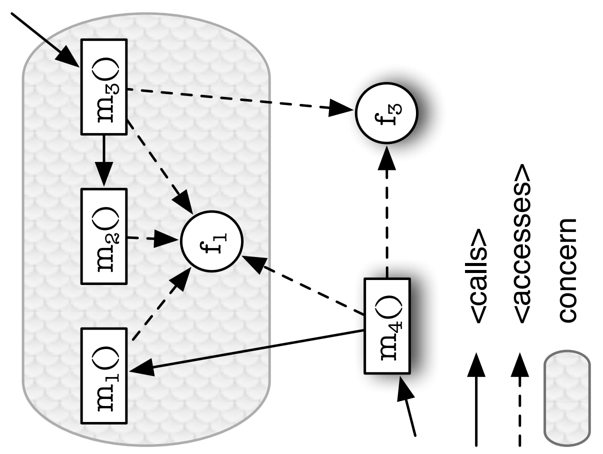
Our hypothesis is that elements in a concern mapping form patterns that are based on their structure within the code base, and that they constitute a stronger representation than just the names of the program elements forming the concern. To be more precise, they form structural relationships, and we can use these to create an intensional rather than extensional (i.e., textual) concern description. Using the structure between concern elements to describe the concern is less prone to changes than a mere list of concern elements. Thus we propose to identify strong structural relationships between concern mapping elements and use this, in a way more abstract, concern description to automatically track the implementation of concerns throughout multiple versions of a software system.

To illustrate the idea, let us look at a very simple example (see Figure 6.1 on page 59). Assume that for a concern C (e.g., the Save-As-PDF feature in a text editor) we have identified via some concern location technique that C's mapping is comprised of three methods m_1, m_2, m_3, as well as a field f_1. Should the three methods m_1, m_2, and m_3 correspond to all methods in the software system accessing field f_1, as illustrated in Figure 6.1(a), then our approach will identify this and produce a representation of the pattern along with the concern mapping. With the details of the pattern associated with the concern mapping, we can check whether the pattern is still valid in later versions of the software system. If it is still valid, nothing happens; should it no longer be valid, as in Figure 6.1(b), we can then automatically detect which new program elements should be added in the concern mapping and if that proposition is taken on, it will make the pattern valid again. Using the example above, where the pattern describes all accessors to field f_1, our approach can be used to automatically detect new program elements which also access f_1; they may now also play a role in concern C's implementation within the program. In this example, method m_4 would now be included since it is also an accessor of field f_1, as illustrated in Figure 6.1(b).

Our idea is based on the assumption that program elements that implement a high-level concern share enough similarities and commonalities for us to detect patterns,



(a) Relationship within Code in Initial Version



(b) Relationship within Code a Few Versions Later

Figure 6.1: Example of Program Element Relationships in Subsequent Software Versions

which in turn allow us to track the implementation of concerns in evolving software systems. In Chapter 7 we discuss the use of an inference engine to identify structural patterns between program elements forming a concern mapping, in order to track evolving concerns in source code. Additionally, we will present how our approach for inferring and checking patterns in concern mappings can be realised and put in action.

6.2 Representing Concerns in Code

A common goal in software engineering is the separation of concerns, resulting in a clean modularisation. However, concerns' implementations often end up scattered across the rest of the program, whether this is due to poor design, or to a system that is too complex to neatly decompose the different functionalities that build the software system. This scattering of concerns, or cross-cutting concerns, is a pressing matter in software engineering, and many techniques have been developed to deal with it. These range from identifying cross-cutting concerns to keeping a record of the code that implements a given concern.

The idea of explicitly representing the scattered code implementing a concern goes back to Soloway et al., who looked at how the software process could be facilitated. They found that conceptual representation methods have a positive impact on the software development process, and propose the writing of cross-referenced textual documentation in the form of annotations to code [46]. More recently, proposed techniques have focussed on concern representations that allow a higher level of automatic analysis. For example, Griswold et al. [26] specify parts of the code belonging to a concern via regular expressions, and developed Aspect Browser around the map metaphor which allows one to manage scattered concern implementations effectively.

We base our approach to representing and modelling concerns in code on the techniques developed in both, the FEAT [43] and ConcernMapper [44] tools, which are discussed in more detail as part of presenting the realisation of our approach in Chapter 7. The remainder of this chapter describes our new approach of intension inference, representation, and tracking of concerns via intensions, combined with the confidence of intensions. We will also discuss how this new idea fits in with existing work in the area of feature location and tracking. This forms the background to the structural inference system which we propose, and whose effectiveness will be evaluated, in Chapter 7.

6.2.1 Concern Modelling

Essentially, a concern mapping consists of an association between a high-level concern and a set of source code elements. Such a set of elements can be discovered in a number of ways, including through manual inspection of the code or through more sophisticated reverse-engineering techniques.

The simplest way to represent a concern's implementation is to record the elements of source code that correspond to its implementation. For example, if a PRINTING feature in a text editor is found to be implemented by four methods $m_1 - m_4$, we would simply record the full signatures of the four methods as associated with PRINTING. In this case we consider the concern to be represented *extensionally*, and the list of elements associated with the concern is its *extension*. This simple scheme is the one used by the ConcernMapper concern modeling tool [44]. The major advantage of representing concerns extensionally is the simplicity of the scheme: one can associate source code elements with a concern simply by listing them. In situations where automatic feature location techniques are used, it is thus possible to take the output of the technique as the concern mapping. On the other hand, the main weakness of the extensional technique is that it is not resilient to evolutionary changes: modifications to the source code (such as refactorings) will easily invalidate the mapping. For example, renamed elements will not be found in later versions of the system.

One can also model concerns *intensionally*.[9] In this case, elements relating to a concern are not listed explicitly but through high-level constructs, called *intensions*, that describe their characteristics. For example, if all of the methods that access field f_1 are involved in the implementation of PRINTING, the intension `all accessors of` f_1 would be recorded. The use of intensions to represent concerns in source code is supported by a number of concern modelling tools, including CME [47], IntensiVE [39], AspectBrowser [26], and FEAT [43]. The advantage of specifying concerns intensionally is that this scheme is more tolerant to evolutionary changes. For instance, in the case described above, if accessors of field f_1 are added or removed as the code evolves, the changes will automatically be reflected in the concern model. In particular, the FEAT tool implements a model of concerns that combines both intensions and extensions, a technique that has been demonstrated to be robust to many types of evolutionary change [41, 43]. Unfortunately, representing concerns intensionally requires more effort from developers, and intensional descriptions are not produced by feature location techniques. We have thus developed a technique to infer intensions automatically based on a given extension and a set of *intension templates* that represent structural patterns among the elements in the extension.

6.2.2 Intension Inference for Structural Relationships

Object-oriented programming, the state-of-the-art in software development these days, encapsulates functionality in classes, methods etc. to provide well-structured code. Thus, we are safe to assume that program code is comprised of *program elements* such as methods and fields. However, these program elements do not float unrelated in space, but form *relationships* that can be purely structural. For instance, given four methods $m_1 - m_4$ and a field f_1, the following situations could occur:

- Methods $m_1 - m_4$ are all the methods that access field f_1.
- Methods $m_2 - m_4$ are all the methods that override m_1 (or any combination thereof).
- Methods $m_1 - m_4$ are all the callers of a fifth method m_5.
- Methods $m_1 - m_4$ and field f_1 constitute all the members of class C.

[9]We use the term "intension" in the sense of Eden and Kazman, to denote a structure that can "range over an unbounded domain" [15, p.150]. See also previous work on the question for a more in-depth discussion of the role of intensions for concern traceability [41, 43].

All the above examples can be encoded as intensions that apply over a certain set of elements. We have used seven *intension templates* for Java systems:

- callerOf a method in the program that calls another method
- calledBy a method in the program that is called by another method
- accessorOf a method in the program that accesses (reads or writes) a field
- accessedBy a field in the program that is accessed (read or written) by a method
- declaredBy a method or field in the program that is declared by a class
- implements a method in the program that implements a method declared in an interface or abstract class
- overrides a method in the program that overrides another method

Of course, some of these relationships are each other's inverse, e.g., callerOf and calledBy, or accessorOf and accessedBy.

The program consisting of its program elements and their relationships can also be seen as graph $G^{\mathcal{L}}$, where each program element e is represented by a different type of node $lin\mathcal{L}$ in the graph $G^{\mathcal{L}}$, and the above relationships are expressed by an edge between its participating nodes (program elements e). The type of node is depending on the kind of program element (e.g., method, field).

Figure 6.2 on page 63 shows a simple example graph, without any **implements** or **overrides** relationships. Rectangular nodes represent methods which are annotated with numbers to distinguish them, round nodes represent fields, also distinguished by numbers. The dot-end of an edge represents the "active" partner of a relationship, i.e., the caller or accessor of another program element; the straight end of an edge represents the "passive" partner of a relationship, i.e., the callee or the accessed field. In this example graph, we can see that, e.g., method 9 is **callerOf** method 8 but also **accessorOf** field 3. However, it is also true that method 8 is **calledBy** method 9, and **accessorOf** field 3 are methods 8 and 9.

Let us now assume that the following elements comprise a description of feature **A**: methods 2 – 5 and fields 1 and 2 (the shaded area in Figure 6.2). If a developer needs to find the implementation of this particular feature in the code, she just looks at that description and can navigate the program code using it. However, over time, the program will evolve, things will get added, deleted, rewritten, renamed, etc. If a developer now wants to find the feature in the code again, maybe because she needs to change something, or because she simply wants to understand how it is implemented, she will quite possibly find the description to be obsolete. As the feature description is *extensional*, a simple list of program elements' names, it is not sustainable over time. However, if we use for example the structural relationships defined above to describe the feature *intensionally*, we might be better able to track and reconstruct a feature's description over time and after changes.

In the example, that would mean that we could describe part of the feature by saying that "**accessorOf** field 1 are methods 2 and 3". If we stored this as the *intension*

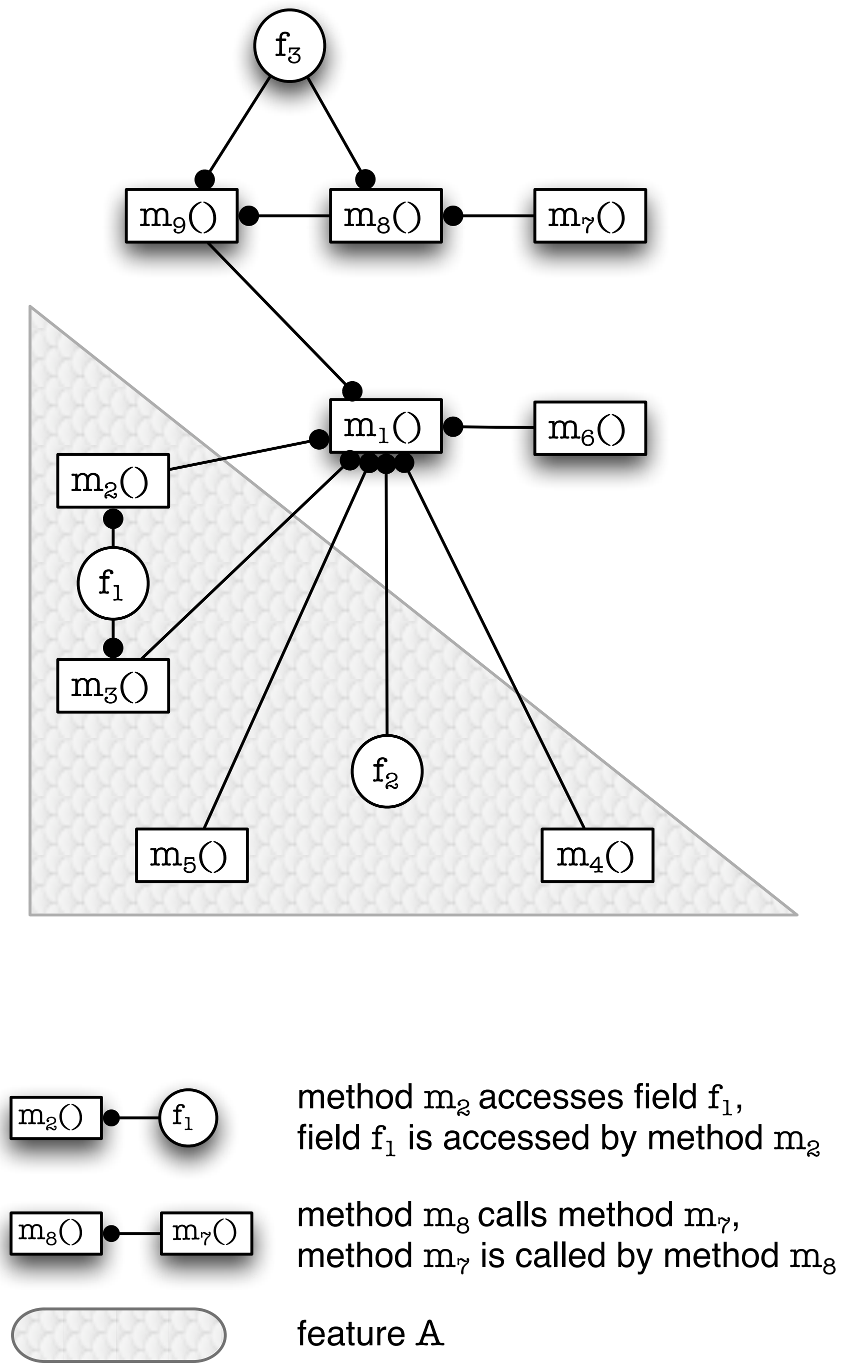

f₃
m₉()
m₈()
m₇()
m₁()
m₆()
m₂()
f₁
m₃()
f₂
m₅()
m₄()
m₂()
f₁
method m₂ accesses field f₁,
field f₁ is accessed by method m₂
m₈()
m₇()
method m₈ calls method m₇,
method m₇ is called by method m₈
feature A

Figure 6.2: Example Graph

"**accessorsOf** field 1", rather than the current extension (methods 2 – 3), we would be able to reconstruct the feature's implementation at a later stage, when maybe method 2 has been renamed, or an additional method has been introduced that also accesses field 1: we just need to calculate the extension of the intension, i.e., to produce all methods for which the proposition that they access field 1 is true. This is like Excel updating inter-cell references after a row/column insertion.

We can actually define the structural relationships on the graph. In general, all edges of the same type from a node to nodes of the same type form/represent an intension. In Figure 6.2 this is for example true for all edges going from the method 1; they are all of the same type (method calls another method), and they all go to the same type of node (method node).

6.2.3 Intension Inference for Textual Similarity

Developers in object-oriented programming do not only encapsulate functionality in program elements such as methods and classes, they also follow so-called *naming conventions*. Naming conventions are rules for picking the sequence of characters or words to be used for the identifiers that denote methods, variables, etc. in a program's source code. Allowing developers to pick any sequence of characters to name program elements will not just cause problems for the person who didn't write the code herself and has to work with it. Following naming conventions decreases the time and effort spent on reading and understanding program code, and improves its appearance.

While the user of software does neither see nor has any benefit from developers following naming conventions, subsequent generations of programmers working on the software system, extending or modifying the code, will have a significantly easier job understanding what the implementation does, and thus will have less trouble fixing bugs or adding new features. A simple example can illustrate this. While `x = y * z;` is syntactically correct in Java, it does not disclose its intent within a program's code. We just know that it multiplies two values and assigns the result to a third variable. However, in `paymentDue = numberOfSupervisionsGiven * supervisionRate;` the chosen identifier names imply the meaning of this bit of code while doing exactly the same from a calculation point of view: the money earned by supervising stundents is calculated as the product of the number of supervisions given and the payment rate for supervisions.

As in this example, naming conventions often also refelct connections of program elements that are all part of a bigger goal, such as implementing a specific functionality and suggest how a developer thinks the program elements fit together. We can even go as far as to assume that the names of methods implementing, e.g., a feature that has to do with exporting into a PDF file, will contain one if not both of the word stems "PDF" and "export".

With this knowledge we can assume that there are more than just structural relationships hidden between program elements that implement a specific feature or functionality. We can use this to define another intension template

- textuallySimilar a program element in the code whose identifier (name) is textually similar to the identifier of another program element.

The term "textually similar" is slightly vague on purpose. The realisation of this intension template depends on the programming language and naming conventions used in the project one wants to analyse, as well as on how strict one wants the textual similarity to be. For example, it can require the exact match of nouns or verbs used for identifiers (e.g., if one includes the word "exporting", identifiers that only include "export" are not considered textually similar), or it can look for matching word stems (e.g., it only needs to share the word stem "export", which would then also match "exporting", "exports", "exporter" etc.). One could go even further and allow synonyms of words to be considered for textual similarity, though this requires the incorporation of dictionaries and not just use the words (word stems) found in the source code of the software system.

6.2.4 Mathematical Notion

While the representation of structural relationships using a graph is very visual, all the previously mentioned relationships can be described by binary relations. Let, e.g., $\textbf{callerOf} = \{(m_i, m_j) | m_i \text{ calls } m_j\}$, $\textbf{callerOf} \subseteq M \times M$, where M denotes all methods in a program and $m_i, m_j \in M$. Similarly, all other relationships can be described as the following binary relations, where M all methods in a program, F all fields in a program, and C all classes in a program:

$$\begin{aligned}
&\textbf{callerOf} = \{(m_i, m_j) \mid m_i \text{ calls } m_j\},\\
&\quad \textbf{callerOf} \subseteq M \times M, m_i, m_j \in M
\end{aligned}$$

$$\begin{aligned}
&\textbf{calledBy} = \{(m_j, m_i) \mid m_j \text{ is called by } m_i\},\\
&\quad \textbf{calledBy} \subseteq M \times M, m_i, m_j \in M
\end{aligned}$$

$$\begin{aligned}
&\textbf{accessorOf} = \{(m_i, f_n) \mid m_i \text{ accesses } f_n\},\\
&\quad \textbf{accessorOf} \subseteq M \times F, m_i \in M, f_n \in F
\end{aligned}$$

$$\begin{aligned}
&\textbf{accessedBy} = \{(f_n, m_i) \mid f_n \text{ is accessed by } m_i\},\\
&\quad \textbf{accessedBy} \subseteq F \times M, f_n \in F, m_i \in M
\end{aligned}$$

$$\begin{aligned}
&\textbf{declaredBy} = \{(e, c) \mid e \text{ is declared by } c\},\\
&\quad \textbf{declaredBy} \subseteq (F \cup M) \times C, e \in F \cup M, c \in C
\end{aligned}$$

$$\begin{aligned}
&\textbf{implements} = \{(m_i, m_j) \mid m_i \text{ implements } m_j\},\\
&\quad \textbf{implements} \subseteq M \times M, m_i, m_j \in M
\end{aligned}$$

$$\begin{aligned}
&\textbf{overrides} = \{(m_i, m_j) \mid m_i \text{ overrides } m_j\},\\
&\quad \textbf{overrides} \subseteq M \times M, m_i, m_j \in M
\end{aligned}$$

$$\begin{aligned}
&\textbf{textuallySimilar} =\\
&\quad = \{(e_i, e_j) \mid e_i\text{'s identifier is textually similar to } e_j\text{'s identifier}\},\\
&\quad \textbf{textuallySimilar} \subseteq (F \cup M \cup C) \times (F \cup M \cup C), e_i, e_j \in (F \cup M \cup C)
\end{aligned}$$

These relations formalise our idea of describing a feature intensionally. Based on the principle of currying functions in the lambda calculus, i.e., fixing some arguments in order to get a function of the remaining arguments, we can "curry" the relations, or partially apply them. The results can be joined into an intensional concern description, which describes an up-to-date set of program elements for which the half-applied relation is true (as we have binary relations here), even in later versions of the program when elements have been deleted, renamed, or newly introduced.

Taking the example in Figure 6.2, we have the following partially applied relations, the union of which are the intensional concern description: **accessorOf**(f_1), **accessorOf**(f_2), **accessedBy**(m_2), **accessedBy**(m_3), and **calledBy**(m_1). However, when actually looking at the sets that these partially applied relations describe, we see that while the resulting sets from the first four are fully a part of the concern, the last contains two elements (namely m_6 and m_9), that are not in the concern (the shaded area). What that means, and the possibilities to address or even use it, will be discussed in the next section.

6.2.5 Confidence

In practice, it is doubtful that we will find intensions that apply completely, i.e., that describe program elements that are all comprised within the concern of interest. Therefore, we need a way to express such incomplete intensions within concerns. We call the degree with which an intension applies its *confidence*. It is a measure that expresses how confident we are that the intension is a valid description of a concern.

Given a concern mapping $\mathcal{C}$, an intension $\mathcal{I}(e)$ based on program element e, and its extension $\mathsf{extension}(\mathcal{I}(e))$ the extension corresponding to the intension $\mathcal{I}(e)$, we define the confidence of intension $\mathcal{I}(e)$ to be the proportion of concern elements $c \in \mathcal{C}$ that are in the extension of $\mathcal{I}(e)$:

$$\mathbf{confidence}(\mathcal{I}(\mathbf{e})) = \frac{|\mathcal{C} \cap \mathsf{extension}(\mathcal{I}(e))|}{|\mathsf{extension}(\mathcal{I}(e))|}$$

As example to illustrate this, Figure 6.3 on page 67 shows an extensional concern description on the right, and the relationships of the program elements on the left. Field f_1 is accessed by four methods $m_1 - m_4$ but only m_1 through m_3 are in concern mapping $\mathcal{C}$. Thus we would say that the confidence of intension $\mathsf{accessorOf}(f_1)$ is 3/4 or 75%. This measure provides us with a way to discriminate between intensions depending on whether their confidence exceeds a certain (adjustable) threshold. It enables us to deal with mistakes developers might make when documenting a concern description extensionally (e.g., by missing a program element), or simply to address the fact of life that there is an exception to every rule. This also applies here where we want to use rules to describe program behaviour but sometimes, while the rule found is a valid intensional description, it will not apply completely. We will see the value of this in the following chapter.

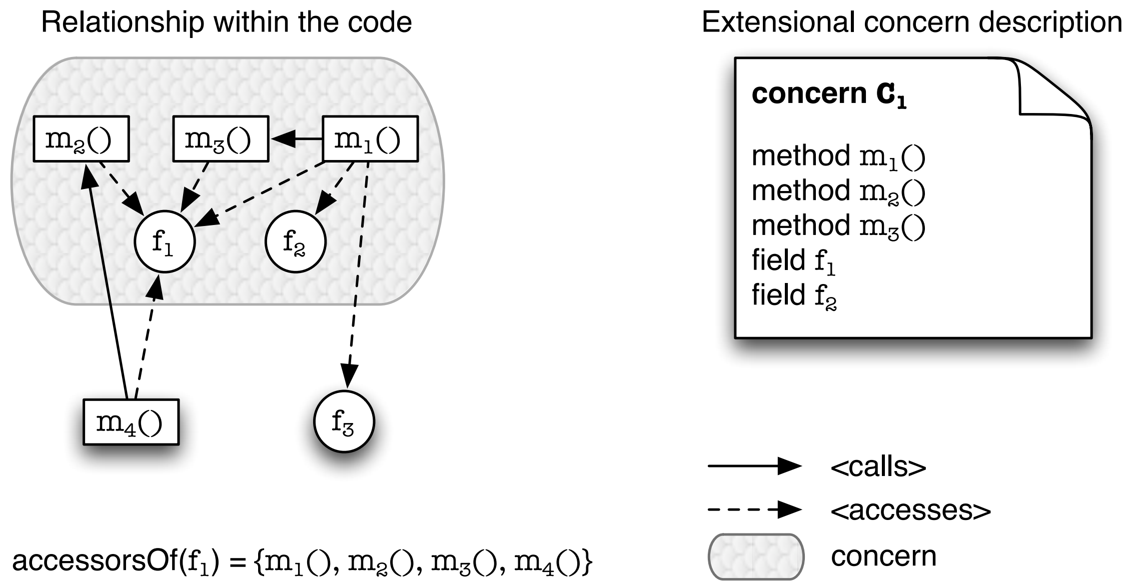


Figure 6.3: Example Concern and Relationships of its Elements in Code

"The important thing is not to stop questioning. Curiosity has its own reason for existing."

Albert Einstein

7 The ISIS4J System

ISIS4J (Implicit Structure Inference System for Java) is a tool implementing our ideas from Chapter 6 about how to infer and apply descriptions of the *implicit structure* of a concern.

The implicit structure is defined as the set of intensions that can be inferred for a concern at a given level of confidence. The given concern is mapped in the extensional form offered by ConcernMapper [44]. A concern mapping describes a feature or an aspect of the software system being analysed by listing all program elements that are involved in the implementation of the feature or aspect.

ISIS4J is implemented as a plug-in for the Eclipse platform[10]. One reason for this is, that ConcernMapper is also realised as an Eclipse plug-in, so that utilising its functionality is easy. The inference process is realised using a business rule management system[11] with a forward chaining inference-based rule engine[12] using the so-called *Rete algorithm* [21]. The Rete algorithm is used commonly to implement matching functionality within pattern-matching engines that employ a match-resolve-act cycle to support inferencing, just as in our problem.

Figure 7.1 on page 70 presents an overview of ISIS4J, tailored for our example in the following. The four main steps, numbered 1 to 4 in the figure are

1. identify list of program elements that implement a given feature,
2. infer implicit structure of the feature by producing a collection of intensions,
3. apply intensions to remove items that no longer exist and add new elements relevant to the feature's implementation, and

[10]`http://www.eclipse.org/`

[11]`drools.org/drools-3.0`

[12]`http://www.jboss.com/products/rules/`

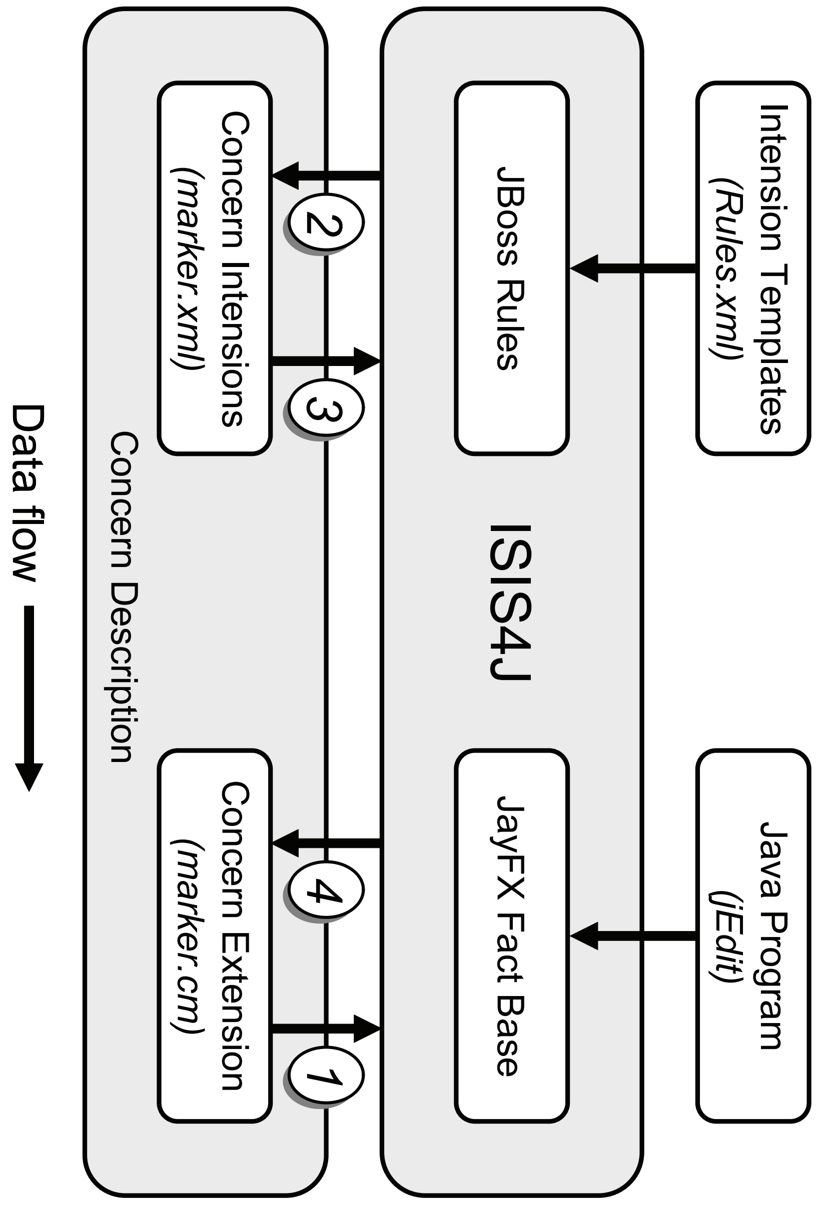


Figure 7.1: ISIS4J Overview

4. provide new list of program elements currently relevant for the feature's implementation.

Before presenting the details of the ISIS4J implementation, a brief overview of ConcernMapper is provided. Then we describe an example concern and illustrate how ISIS4J can help maintain the knowledge of the concern's implementation as the system's code evolves. Following that, we focus on evaluating several research questions. Does the implementation of a feature have an implicit structure that can be described using intensions as defined by us? How useful is the representation as implicit structure for tracking the possibly changing nature of a feature's implementation?

7.1 ConcernMapper

It is established that separation of concerns is a tricky challenge that cannot always be achieved completely. When adapting or extending a software system, developers often have to find the involved code, scattered across many files. While programming environments such as Eclipse provide a certain amount of help navigating and locating code, it goes beyond their capabilities to guide the developer through changes that are related in a feature's context but are not necessarily clearly co-located.

ConcernMapper [44] is a concern management tool for Java software projects, implemented as an Eclipse plugin. It is thus easily extensible, and we used it as the basis for our analysis needs and extended it to record data about features' implementations. ConcernMapper's goal is to support daily software development by offering an easy-to-use drag-and-drop approach to managing a concern's or feature's implementation without altering or touching the actual code. The developer can maintain a simple list of program elements that are involved in a concern's implementation, and this description can be stored for future use. Thus, ConcernMapper provides a means to have an *extensional* concern representation, also referred to as *concern model* by the developers of ConcernMapper and its underlying theory.

Let us describe a typical usage scenario for the tool ConcernMapper within Eclipse. A programmer has to modify an existing feature F in a software system. Assume the worst case, that the programmer is new to the development team. The first thing she has to do is understand how F is implemented. She can search for keywords that describe the feature's functionality in Eclipse, using different views provided, such as the Package Explorer, or Search. Browsing the code based on the various search results, the programmer will try to understand how the feature is implemented. Then, she will create within ConcernMapper a concern model C consisting of all elements that she considers relevant for F's implementation. While the programmer continues to browse the code base, she will be informed, e.g., when looking at classes of her project, which program elements (e.g., methods) already belong to her concern description in ConcernMapper. Should she find that the feature she is aiming to understand consists of several sub-features, the programmer can decide to create new concerns, moving elements around the different concern descriptions by simple drag-and-drop. Once the programmer thinks she has understood how the feature F is implemented, she can save the concern model C and any sub-concern models she built for future reference.

When the programmer continues to work on the modifications she needs to do on F's implementation, she merely loads C in ConcernMapper, and by selecting a program element listed in the model, she can directly access the underlying code of that element. So, the concern representation provided by ConcernMapper is extensional since it lists all program elements involved in a concern's implementation. It is stored internally in a file with the ending .cm (e.g., foo.cm), while we have chosen to store the corresponding intensional representation in XML format (e.g., foo.xml).

7.2 Typical usage scenario for ISIS4J

We base our example on the jEdit open-source text editor[13]. jEdit includes the MARKER feature which allows users to bookmark certain lines in a file, i.e. for our purposes a set of lines of code. It is a standard jEdit example feature and has been used previously by other researchers when illustrating or evaluating their work. Imagine that a developer has been assigned the task of modifying the MARKER feature. For that, she needs to identify those parts in the code that implement the feature. Thus, the developer will consult any available documentation, browse the code and use different means at her disposal to find the methods and fields that are associated with the MARKER feature. She will then complete the modification task and commit the changes. The program elements that make up the feature's implementation can then simply be obtained as the list of methods changed when committing, or could be marked explicitly during the task of identifying how the feature is implemented by the developer using a tool like ConcernMapper (described above), saved in a file called `marker.cm`.

The developer then runs ISIS4J to infer the implicit structure of the marked elements, producing a collection of intensions that are saved in the `marker.xml` file. This part of the process is represented by arrows 1 and 2 in Figure 7.1. The intensions describe the implicit structure that part (or all) of the concern mapping elements have within the source code. An excerpt from the example file `marker.xml` is shown in Figure 7.2 on page 73. In that example[14], the rule produces all methods that are declared by the class `org.gjt.sp.jedit.Marker`; the `rule name "rDeclared-By_org.gjt.sp.jedit.Marker_in_Marker"` indicates this. The interesting part is the right-hand side (wrapped in between `<rhs>` and `</rhs>`). There, the rule adds the element `Marker` in conjunction with the intension relationship `relation.DECLARES`. This means that everything that is declared in the class Marker will be returned by this intension, also referred to as 'rule'.

A few software versions later, the developer needs to perform another modification task on the same feature. To save herself the tedious work of browsing the code yet another time to identify the program elements currently involved in the implementation of the MARKER feature, the developer uses ISIS4J. She runs ISIS4J, using the marker intensions set `marker.xml` (arrow 3) and the original concern mapping file `marker.cm` (arrow 1). Based on this input, ISIS4J removes the program elements listed in the concern mapping file that no longer exist (so-called inconsistencies, e.g., methods or

[13]`http://www.jedit.org/`

[14]As previously mentioned, this is our chosen way of expressing intensions, see Chapter 6.

```
...
<rule name="rDeclaredBy_org.gjt.sp.jedit.Marker_in_Marker">
<!-- All methods declared by the given class
      are part of the concern model -->
  <lhs>
      <column identifier="javadb" object-type="JavaDB"/>
      <column identifier="relation" object-type="Relation"/>
      <column identifier="cmw" object-type="ConcernModelWrapper"/>
      <column identifier="rw" object-type="RuleWriter"/>
      <column identifier="concern" object-type="String"/>
      <column identifier="type" object-type="ClassElement">
        <literal evaluator="==" field-name="id"
          value="org.gjt.sp.jedit.Marker"/>
      </column>
  </lhs>
  <rhs>cmw.addElements( "Marker",
    "rDeclaredBy_org.gjt.sp.jedit.Marker_in_Marker",
    getRange( javadb, relation.DECLARES, type), 100 );
  </rhs>
</rule>
...
```

Figure 7.2: Example for an intension in XML file `marker.xml`

fields that were deleted). Then, ISIS4J adds the new elements that are found by applying the intensions (e.g., a new method accessing a field identified by the accessorOf intension). Applying an intension (or a rule) refers to executing an applicable rule on an input, similar to how a mathematical function can be applied to a value. The concern mapping file `marker.cm` is updated and the developer can now check the newly discovered elements in the new concern mapping file (arrow 4) for relevance, and then proceed with the planned modification task.

7.2.1 Implementation

The previously described scenario is completely supported by our ISIS4J prototype, described as follows:

Explicit Concern Model. The main input of ISIS4J is a concern mapping file (e.g., `marker.cm`) holding extensions created using the ConcernMapper plug-in. In our current implementation, the user creates the initial definition of the concern of interest by explicitly selecting Java methods and fields from the current version of a Java project and adding them to the concern model (a set of program elements) by drag-and-drop in the ConcernMapper view. A concern mapping file for our Marker concern can be seen in Figure 7.3 on page 74. For example, the mapping includes methods such as `getMarkers()` from the `class org.gjt.sp.jedit.Buffer` and `removePosition()`

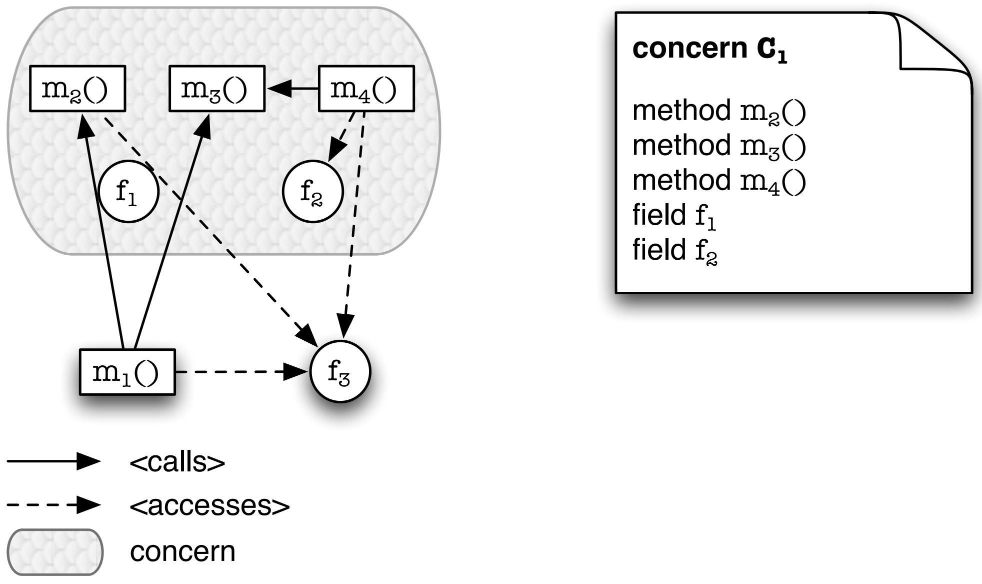
```
<element degree="100" id="org.gjt.sp.jedit.Buffer.getMarker(C)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Buffer.markers" type="field"/>
<element degree="100" id="org.gjt.sp.jedit.textarea.JEditTextArea.goToNextMarker(Z)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Buffer.removeMarker(I)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.GUIUtilities.loadMenu(java.lang.String)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.textarea.JEditTextArea.swapMarkerAndCaret(C)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Buffer.getMarkerAtLine(I)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.textarea.JEditTextArea.goToMarker(C,Z)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Buffer.parseBufferLocalProperties(java.lang.String)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Buffer.removeAllMarkers()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.jEdit.openFiles(org.gjt.sp.jedit.View,
   java.lang.String,[java.lang.String)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.textarea.JEditTextArea.addMarker()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.io.BufferIORequest.&lt;init&gt;
   (I,org.gjt.sp.jedit.View,org.gjt.sp.jedit.Buffer,
   java.lang.Object,org.gjt.sp.jedit.io.VFS,java.lang.String)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.gui.MarkersMenu.&lt;init&gt;()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.getShortcut()" type="method"/></concern></model>
```

Figure 7.3: Example ConcernMapper file `marker.cm`

```
<?xml version="1.0" encoding="UTF-8"?>
<model><concern name="Marker">
<element degree="100" id="org.gjt.sp.jedit.Buffer.getMarkers()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.io.BufferIORequest
   .writeMarkers(org.gjt.sp.jedit.Buffer,java.io.OutputStream)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.position" type="field"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.buffer" type="field"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.pos" type="field"/>
<element degree="100" id="org.gjt.sp.jedit.gui.MarkersMenu
   .setPopupMenuVisible(Z)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Buffer.addMarker(C,I)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.textarea.JEditTextArea
   .goToPrevMarker(Z)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.io.BufferIORequest.save()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.removePosition()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.jEdit
   .gotoMarker(org.gjt.sp.jedit.View,org.gjt.sp.jedit.Buffer,java.lang.String)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.getPosition()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Buffer
   .&lt;init&gt;(java.lang.String,Z,Z,java.util.Hashtable)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.shortcut" type="field"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.createPosition()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.io.BufferIORequest.load()" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Marker
   .&lt;init&gt;(org.gjt.sp.jedit.Buffer,C,I)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.io.BufferIORequest.markersPath" type="field"/>
<element degree="100" id="org.gjt.sp.jedit.Marker.setShortcut(C)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.Buffer.addOrRemoveMarker(C,I)" type="method"/>
<element degree="100" id="org.gjt.sp.jedit.io.BufferIORequest
   .readMarkers(org.gjt.sp.jedit.Buffer,java.io.InputStream)" type="method"/>
```

from the `class org.gjt.sp.jedit.Marker`, as well as fields such as `shortcut` from the `class org.gjt.sp.jedit.Marker`.

In the following, we use a simple example to illustrate the intension inference steps. Figure 7.4 represents that for code elements constituting a concern, and their relationship with other code elements.



(a) Relationship within Code (b) Extensional Concern Description

Figure 7.4: Example Code and Concern

Data Mining and Rule Engine. Once a concern of interest has been defined, ISIS4J uses the JayFX[15] Eclipse plug-in to create a fact base containing all structural relationships such as method calls, inheritance relationships, and field accesses in the source code of the project. JayFX uses class hierarchy analysis (CHA) [14] to conservatively infer the targets of polymorphic calls. ISIS4J then infers the implicit structure of the concern by checking its elements against the fact base. ISIS4J performs this data mining operation using the JBoss Rules engine[16] and a set of rule templates that we defined in the Drools 3.0 XML language[17] (e.g., `Rules.xml`, Figure 7.1; although this is an XML file, it is not a "collection of intensions").

JBoss Rules is a reasoning engine that includes a *forward chaining* rule engine based on Drools. Forward chaining is one way of reasoning using inference rules (as in our case). It starts with the available data, and uses inference rules to extract more data until a goal is reached. The inference rules, consisting of an if-clause (antecedent) and a then-clause (consequent), are searched for those where the antecedent is true. Once found, the rule engine infers the consequent, which leads to the addition of new

[15]`http://www.cs.mcgill.ca/$\sim$swevo/jayfx/`
[16]`http://www.jboss.com/products/rules/`
[17]`drools.org/drools-3.0`

```
<rule name="inferDeclaredBy">
<!-- All methods declared by the given class
     are part of the given concern -->
  <condition>
    <column id="concern" type="Concern"/>
    <column id="class" type="ClassElement"/>
    <eval>
      checkConfidence(Relation.DECLARES,class,concern)
    </eval>
  </condition>
  <consequence>
     RuleWriter.createIntension("declaredBy",
        Relation.DECLARES,class);
  </consequence>
</rule>
```

Figure 7.5: Example Rule Template

information to its data. This process is iterated until the set goal is reached, which means it is basically what is commonly also known as "worklist" algorithm.

Drools is a Java implementation of a rules engine, specifically fitted for Java. It is based on Charles Forgy's Rete algorithm [21], and adapting Rete to an object-oriented interface allows us more naturally to express business rules with regards to business objects. Drools allows different language implementations, thus rules can be written in Java, Python, etc. All of this makes it a good fit for our purposes. Rule templates expressed in Drools correspond to our intension templates as described in Section 6.2.2.

For example, as illustrated by Figure 7.5, there is one rule template that is responsible for discovering whether all methods of a given class are contained in the user-defined concern. For a class (`type="ClassElement"`) in a concern (`type="Concern"`), we check what proportion of methods in this class are part of the concern (`<eval>...</eval>`). That is the `<condition>`. If such a situation occurs, that rule template would then generate the declaredBy intension defined in Section 6.2.2, which is stated in the `<consequence>` part of the rule template. In other words, ISIS4J goes through all Java elements (interfaces, classes, methods and fields) in the project and, for each element, it computes each one of the seven relationships: callersOf, calledBy, accessorsOf, accessedBy, overrides, implements, and declaredBy. It then generates an intension (e.g., declaredBy(`Buffer`)) associated with this particular element if a sufficient proportion of the relationship extension is in the concern (i.e., if the intension meets a predefined confidence threshold; see Section 6.2.5).

Figure 7.6 on page 78 illustrates a step-by-step inference of an intensional concern description for the example from Figure 7.4. Methods $\mathtt{m}_2$() and $\mathtt{m}_3$(), elements of the concern description, are called by method $\mathtt{m}_1$(). $\mathtt{m}_3$() is also called by method $\mathtt{m}_4$(). These relationships can be expressed as calledBy intensions: calledBy($\mathtt{m}_1$()) and calledBy($\mathtt{m}_4$()). Furthermore, methods $\mathtt{m}_2$() and $\mathtt{m}_4$(), both part of the concern, access the field $\mathtt{f}_3$, and method $\mathtt{m}_4$() also accesses the field $\mathtt{m}_2$. Accordingly, we can express these relationships by the intensions accessorsOf($\mathtt{f}_2$) and accessorsOf($\mathtt{f}_3$). Similarly, we can infere the intension accessedBy($\mathtt{m}_4$()), which describes the relationship that method $\mathtt{m}_4$() has with the fields $\mathtt{f}_2$ and $\mathtt{f}_3$.

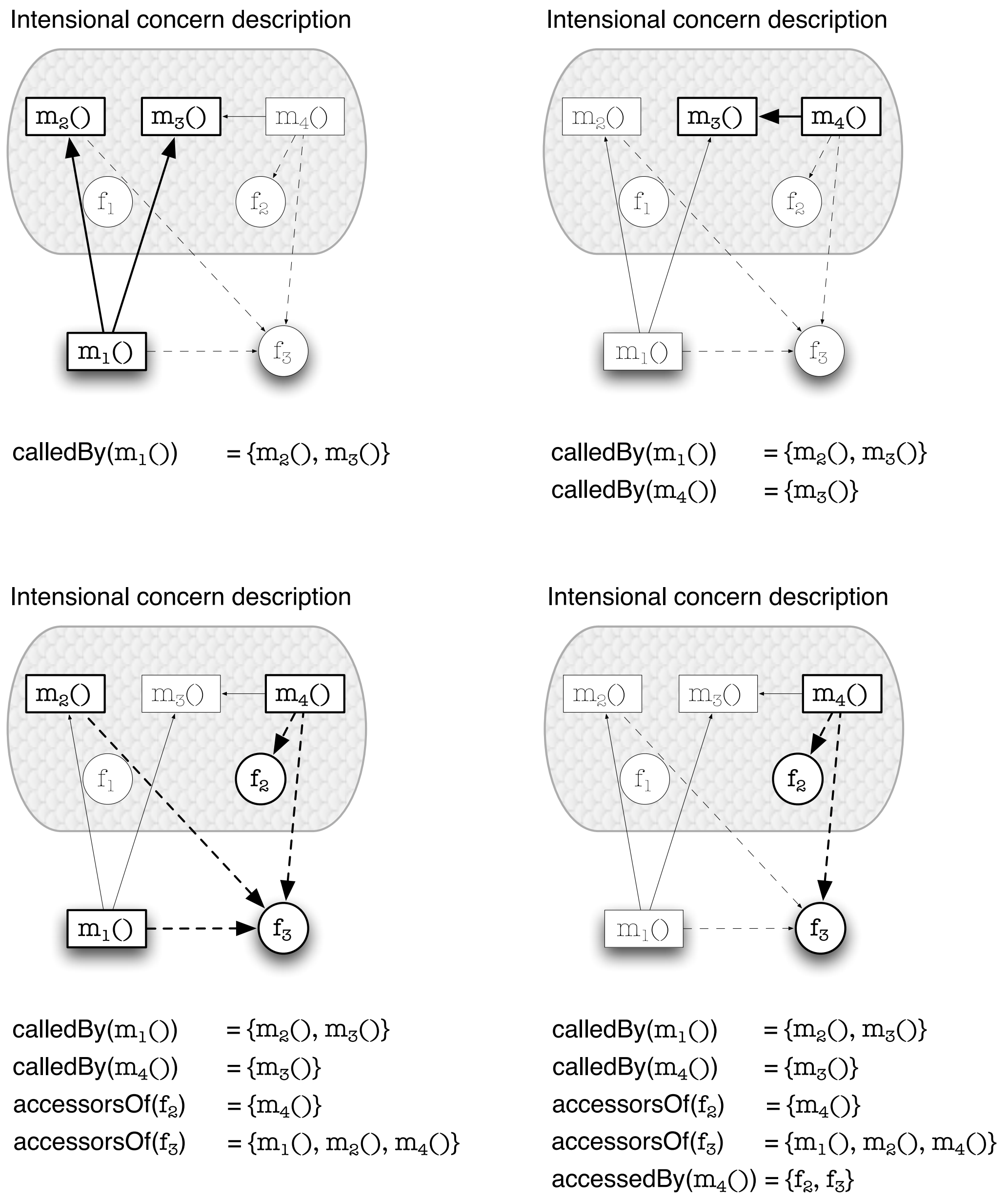

Intensional concern description
m2() m3() m4() f1 f2 m1() f3
calledBy(m1()) = {m2(), m3()}
Intensional concern description
m2() m3() m4() f1 f2 m1() f3
calledBy(m1()) = {m2(), m3()}
calledBy(m4()) = {m3()}
Intensional concern description
m2() m3() m4() f1 f2 m1() f3
calledBy(m1()) = {m2(), m3()}
calledBy(m4()) = {m3()}
accessorsOf(f2) = {m4()}
accessorsOf(f3) = {m1(), m2(), m4()}
Intensional concern description
m2() m3() m4() f1 f2 m1() f3
calledBy(m1()) = {m2(), m3()}
calledBy(m4()) = {m3()}
accessorsOf(f2) = {m4()}
accessorsOf(f3) = {m1(), m2(), m4()}
accessedBy(m4()) = {f2, f3}

Figure 7.6: Example: Inferring Intensional Concern Description

Figure 7.7 on page 79 explains how to calculate the aforementioned confidence of intensions. It uses the example concern $\mathtt{C}_1$ from Figure 7.4 and the intensions inferred for it as shown in Figure 7.6. The confidence of the five inferred intensions calledBy($\mathtt{m}_1$()), calledBy($\mathtt{m}_4$()), accessorsOf($\mathtt{f}_2$), accessorsOf($\mathtt{f}_3$), and accessedBy($\mathtt{m}_4$()) are 100%, 100%, 100%, 66.67%, and 50% respectively. Depending on the chosen confidence threshold,

INFERRED INTENSIONS

calledBy($m_1()$) $= \{m_2(), m_3()\}$
calledBy($m_4()$) $= \{m_3()\}$
accessorsOf(f_2) $= \{m_4()\}$
accessorsOf(f_3) $= \{m_1(), m_2(), m_4()\}$
accessedBy($m_4()$) $= \{f_2, f_3\}$

CALCULATION OF INTENSIONS' CONFIDENCE

$$\textbf{confidence}(\mathcal{I}(e)) = \frac{|\, C_1 \cap \text{extension}(\mathcal{I}(e)) \,|}{|\, \text{extension}(\mathcal{I}(e)) \,|}$$

confidence(calledBy($m_1()$)) =

$$= \frac{|\{m_2(), m_3(), m_4(), f_1, f_2\} \cap \{m_2(), m_3()\}|}{|\{m_2(), m_3()\}|}$$

$$= \frac{|\{m_2(), m_3()\}|}{|\{m_2(), m_3()\}|} = \frac{2}{2} = 1 \triangleq 100\%$$

confidence(calledBy($m_4()$)) =

$$= \frac{|\{m_2(), m_3(), m_4(), f_1, f_2\} \cap \{m_3()\}|}{|\{m_3()\}|}$$

$$= \frac{|\{m_3()\}|}{|\{m_3()\}|} = \frac{1}{1} = 1 \triangleq 100\%$$

confidence(accessorsOf(f_2)) =

$$= \frac{|\{m_2(), m_3(), m_4(), f_1, f_2\} \cap \{m_4()\}|}{|\{m_4()\}|}$$

$$= \frac{|\{m_4()\}|}{|\{m_4()\}|} = \frac{1}{1} = 1 \triangleq 100\%$$

confidence(accessorsOf(f_3)) =

$$= \frac{|\{m_2(), m_3(), m_4(), f_1, f_2\} \cap \{m_1(), m_2(), m_4()\}|}{|\{m_1(), m_2(), m_4()\}|}$$

$$= \frac{|\{m_2(), m_4()\}|}{|\{m_1(), m_2(), m_4()\}|} = \frac{2}{3} = 0.67 \triangleq 66.67\%$$

confidence(accessedBy($m_4()$)) =

$$= \frac{|\{m_2(), m_3(), m_4(), f_1, f_2\} \cap \{f_2, f_3\}|}{|\{f_2, f_3\}|}$$

$$= \frac{|\{f_2\}|}{|\{f_2, f_3\}|} = \frac{1}{2} = 0.5 \triangleq 50\%$$

Figure 7.7: Example: Calculating Intension Confidence

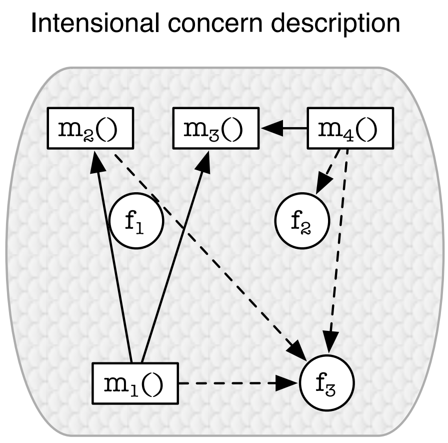
only some of the intensions will be considered strong enough to describe the concern C_1. Should the developer have chosen to accept only perfect confidence, i.e., 100%, only calledBy($m_1()$), calledBy($m_4()$), and accessorsOf(f_2) will be kept. If the confidence threshold is more relaxed, e.g., 60%, then all inferred intensions but accessedBy($m_4()$) are added to the set of intensions describing concern C_1.

Implicit Structure Represented as a Set of Intensions. The output of ISIS4J is an executable rule set describing the inferred intensions in the user-defined concern, represented in the XML syntax of Drools 3.0 primitives like `<condition>` or `<consequence>`. ‘Executable’ here refers to the rule engine iterating through the set of inference rules (intensions), checking for if-clauses evaluating to True, and thus inferring more data. One could also say that we 'apply' the inference rules (intensions). We chose to use rules to express intensions since we already had access to the fact base and the rule engine. In theory, intensions could be expressed by a variety of other mechanisms, such as concern graphs [43] or JQuery queries [28]. It should be noted that, in practice, there might be Java elements in a concern mapping that are not described by any intension. An example can be seen in Figure 7.4; there, according to the extensional concern description, the field $\mathtt{f}_1$ belongs to the concern's implementation, but none of our defined intensions could be used to describe it. Still, the rule set can be executed using the JBoss Rules engine on any version of the Java project to add to the concern all Java elements corresponding to its implicit structure: we refer to this step as *applying the concern's intensions* while the additional elements are the result of the *intension-based expansion*. Obviously, this can only happen if we use a confidence threshold of less than 100%.

As an example, let us assume a confidence threshold of 65%. If we apply the intension $\mathsf{accessorsOf}(\mathtt{f}_3)$, we get $\{\mathtt{m}_1(), \mathtt{m}_2(), \mathtt{m}_4()\}$ as resulting set of program elements. $\mathtt{m}_1()$ is not part of the concern description. Since the intension's confidence is 66.67% our set threshold of 65% is exceeded. Thus we allow the inclusion and use of the intension $\mathsf{accessorsOf}(\mathtt{f}_3)$ into our rule set, as well as the intension-based expansion of the concern $\mathtt{C}_1$ itself. This results in including method $\mathtt{m}_1()$ in the (extensional) concern description $\mathtt{C}_1$, which expands the original $\mathtt{C}_1$.

Customisable Inference. As mentioned in Section 6.2.5, ISIS4J can infer intensions describing the concern structure even if not all elements matched by the intensions are present in the original concern's extension, provided that their confidence reaches a certain threshold. This threshold is expressed as an integer value between 0 and 100 representing the required confidence level in percentage and can be set in the `Rules.xml` file by the user. To recall, `Rules.xml` is a file that contains the definition of all rule templates we use. ISIS4J uses the confidence threshold parameter when it infers a concern's implicit structure. Allowing a threshold under 100% raises the following issue: if ISIS4J—as we call it—completes an intension by adding missing elements to the concern, it is possible that the added elements result in increasing the confidence for other intensions above the threshold, allowing those intensions to be added to the intension set. We thus introduced an option to run ISIS4J until it reaches a fixed point (where new intensions can no longer be inferred). The fixed point iteration can have the positive impact of completing incomplete extensional concern descriptions (see Section 7.2.1).

Figure 7.8 on page 81 illustrates an example of fixed point inference. Let us assume a confidence threshold of 50%. Then all five of the previously discussed intensions constitute the set of intensions describing the concern $\mathtt{C}_1$. The intension-based expansion of concern $\mathtt{C}_1$ results in additionally including method $\mathtt{m}_1()$ as well as field $\mathtt{f}_3$ since all five intensions reach or exceed a confidence of 50%. Now, if we re-assess the concern, we will find two more relationships involving program elements belonging to the concern $\mathtt{C}_1$: method $\mathtt{m}_1()$ accesses field $\mathtt{f}_3$, and method $\mathtt{m}_2()$ accesses field $\mathtt{f}_3$. This means we can infer two more new intensions $\mathsf{accessedBy}(\mathtt{m}_1())$ and $\mathsf{accessedBy}(\mathtt{m}_2())$, highlighted in boldface in Figure 7.8. After adding both intensions to our set, no fur-



calledBy($m_1()$)	= {$m_2()$, $m_3()$}
calledBy($m_4()$)	= {$m_3()$}
accessorsOf(f_2)	= {$m_4()$}
accessorsOf(f_3)	= {$m_1()$, $m_2()$, $m_4()$}
accessedBy($m_4()$)	= {f_2, f_3}
accessedBy($m_1()$)	**= {f_3}**
accessedBy($m_2()$)	**= {f_3}**

Figure 7.8: Example: Fixed Point Inference

ther intensions exceeding the chosen confidence threshold can be inferred, and no more new program elements can be added to our concern description, thus our fixed point iteration terminates. Note, that in this example, the fixed point inference also results in higher confidence values for accessorsOf(f_3) and accessedBy($m_4()$); they rise both to 100% (from 66.67% and 50% respectively).

7.3 Evaluation: Implicit Structure

For ISIS4J 4J to be useful to software developers we need concern mappings to have *latent* intensions. By 'latent' we mean that the program elements in a concern mapping are connected via relationships which are not visible just by looking at the list of program elements that constitute a (extensional) concern (mapping). An example of such a latent intension would be if all functions in the concern mapping access a certain field when executed. We investigated the nature and characteristics of the intensions produced by ISIS4J 4J on a number of benchmark concern mappings independently produced by different subjects on four open-source systems, as part of a previous empirical study of feature location.

Research Questions

We evaluated our assumption that the implementation of concerns has an implicit structure and how useful it is by assessing three specific characteristics of our approach:

1. What is the impact of the confidence threshold on the number of intensions detected?
2. ISIS4J can infer intensions in a single pass, or can repeat this process until a fixed point is reached and no new intensions can be inferred. What are the tradeoffs of using the latter method (see Section 7.2.1)?
3. We defined seven kinds of intension templates. Are all of these equally useful for generating intensions?

7.3.1 Experimental Design

To collect initial evidence allowing us to answer the questions in Section 7.3, we analysed 16 concerns obtained from a previous empirical study of feature location. In this study, different subjects created concern mappings for 16 different concerns in four different systems (Table 7.1 on page 82).

System	**Version**	**LOC**	**Concern Size**
GanttProject	2.0.2	43 246	22, 30, 26, 13
Jajuk	1.2	30 679	14, 18, 17, 11
jBidWatcher	1.0pre6	23 051	20, 20, 10, 19
Freemind	0.8.0	70 435	18, 28, 16, 14

Table 7.1: Characteristics of Target Systems

Our target systems shared the following characteristics: all projects are mature and actively maintained, and their size (in LOC) is well beyond that of toy systems. As for the concern descriptions, they consisted of a paragraph of text written by two investigators as part of a different study. The concern descriptions were created by explaining a feature of the system as mentioned in the user manual, help pages, or user interface of the system.
For each concern, three different subjects were asked to identify the fields and methods that were judged to be the most relevant to the implementation of the concern. The subjects were undergraduate and graduate students with Java programming experience in Eclipse. No method, process, or tool (besides the features of the Eclipse environment) was given them to complete this task. This process resulted in $16 \times 3 = 48$ different mappings. Since we were interested in studying the documentation of relatively large sets of elements corresponding to a concern's implementation, we selected, for each concern, the mapping with the highest number of elements. This resulted in 16 data points corresponding to one mapping for each concern (four in each of our target system). In Table 7.1, the last column lists the cardinality of each of the four mapping

considered for each system. For example, for GanttProject, our four concern mappings included 22, 29, 22, and 13 elements, respectively. The average concern mapping size across our 16 data points is 18.5 elements. Our sample of 16 concern mappings thus represent the fields and methods that would be identified as relevant to a concern as manually determined by a junior programmers.

For each concern mapping, we used ISIS4J to infer intensions and we then completed the concern mapping definition by applying those intensions (i.e., for incomplete but above-threshold intensions, we added the elements missing in the intensions' extension). We automated ISIS4J and parametrised its execution according to two variables.

Fixed Point. We evaluated the impact of reaching a fixed point (see Section 7.2.1) where no more new intensions could be inferred versus inferring intensions only once based on the initial concern mapping.

Threshold. We tested the effect of four different threshold values for the intension confidence of 60%, 75%, 90% and 100%, where intension confidence describes the proportion of program elements forming the concern that are in the extension of the intension.

7.3.2 Results

Table 7.2 on page 84 shows our results. For each concern (Cn.), for different thresholds, with both the single (S) and fixed point (FP) inference options, the table lists the number of intensions detected with ISIS4J and the number of elements added by applying the inferred intensions (in parentheses). The last column of the table provides the median value of each column. The original concern mappings size is reported in Table 7.1 on page 82 (mappings 1-4 for GanttProject, 5-8 for Jajuk, 9-12 for jBidWatcher and 13-16 for Freemind). For example, for concern 1, with a threshold of 75% and using a single inference pass, ISIS4J inferred two intensions, and applying these intensions resulted in adding three elements to the original concern mapping's 22 elements (see Table 7.2). Based on this data, we provide answers to each of our research questions.

Impact of Confidence on Number of Intensions. For a threshold above 90%, almost no intensions were inferred.[18] Our experiment shows that concern mappings produced manually do not tend to follow rigorously regular structural lines. We see that using a more relaxed threshold (of 60%–75%) yields, in many cases, the inference of intensions among the elements of a concern. This observation thus highlights the fact that in order to preserve intensional knowledge about the implementation of a concern, the extension corresponding to latent intension must often be completed.

Fixed Point Inference Trade-Offs. Fixed point inference only made a substantial difference when using a threshold of 60% (above that level, the threshold inhibited the inference of intensions as can be observed by the low number and small difference of inferred intensions with fixed point and single pass inference). With fixed point inference, ISIS4J discovered 59% more intensions than with single pass inference for a total of 110 intensions instead of 69. (The average is respectively 6.9 and 4.3 intensions per concern mapping.) A higher number of intensions per concern mapping is desirable

[18]Results for 90% and 100% are identical.

	Cn.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**Mdn**
60%	**S**	16(10)	3(3)	3(4)	3(1)	0(0)	8(4)	2(1)	2(2)	2(3)	5(5)	0(0)	3(1)	4(4)	12(8)	2(1)	4(0)	3
	FP	25(22)	11(12)	10(4)	3(1)	0(0)	10(5)	2(1)	5(5)	3(5)	5(5)	0(0)	3(1)	4(4)	23(13)	2(1)	4(0)	4
75%	**S**	2(3)	0(0)	1(1)	1(1)	0(0)	4(1)	2(1)	1(1)	0(0)	1(1)	0(0)	2(0)	3(3)	4(2)	0(0)	3(0)	1
	FP	4(4)	0(0)	1(1)	3(1)	0(0)	6(1)	2(1)	1(1)	0(0)	1(1)	0(0)	2(0)	3(3)	6(4)	0(0)	3(0)	1.5
90%	**S**	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	1(0)	0(0)	0(0)	0(0)	0(0)	2(0)	2(0)	2(0)	0(0)	3(0)	0
	FP	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	1(0)	0(0)	0(0)	0(0)	0(0)	2(0)	2(0)	2(0)	0(0)	3(0)	0
100%	**S**	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	1(0)	0(0)	0(0)	0(0)	0(0)	2(0)	2(0)	2(0)	0(0)	3(0)	0
	FP	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	1(0)	0(0)	0(0)	0(0)	0(0)	2(0)	2(0)	2(0)	0(0)	3(0)	0

Table 7.2: Number or Intensions with Single and Fixed Point Inference at Different Confidence

when tracking a concern in an evolving system since ISIS4J can then track more types of change (one per intension).

On average, intension-based expansion performed with fixed point inference made the concern mapping's size grow by 23% while single pass inference made it grow by only 14%. In both cases, 75% of the concern mappings did not grow by more than 25%. Thus, most of the final mapping produced by ISIS4J was representative of the original mapping done by the subjects. This provides evidence that the quality and relevance of concern mappings produced by ISIS4J do not degrade significantly.

For both fixed-point and single-pass inference, intensions described between 46% and 86% of the elements in most of the concern mappings (10 mappings out of 16). In other words, one element out of two was in the extension of at least one intension. This suggests that ISIS4J is capable of inferring an implicit structure representative of most of the concern mapping. Moreover, since changes can only be tracked through intensions, it is vital that most of the concern mapping is described by intensions (as more changes related to the concern will be captured).

Although fixed-point inference produced 59% more intensions, the additional intensions described no more than 16% additional elements. This indicates that the additional intensions mostly described elements that were already in the extension of existing intensions. For example, if we take method m_1 that accesses fields f_1 and f_2, a single pass inference might detect the accessorsOf(f_1) intension while the fixed point inference will detect an additional accessorsOf(f_2) intension: both intensions describe the same element, m_1. Intensions that applied return the same values can be useful to track changes as they are potentially more robust: in our example, even if field f_1 is refactored or deleted, we can still track the accessors of f_2.

In the light of such evidence, we conclude that performing fixed-point inference is desirable as it makes the concern implicit structure potentially more robust to changes while not degrading significantly the quality and relevance of the original concern mapping.

Use of Intension Templates. In all of our sample concerns, neither an overrides intension nor a implements intension was inferred. The number of declaredBy as well as accessedBy intensions inferred varies between 0 and 2 per concern. In most other cases, the distribution over the remaining three intensions, calledBy, callersOf, and accessorsOf, is fairly even. We expected that field accesses and method calls would constitute the most significant part of the inferred intensions, but since concerns can exhibit very different characteristics (e.g. one could use design patterns which typically rely on polymorphism and inheritance), a good diversity of intension templates is still important. In addition, inferring those intensions represents a negligible performance cost when this performed along with the inference of the other intensions.

To summarise, we learnt the following lessons from the evaluation:

1. Manually extracted concern mappings benefit from completion by intension inference at a lower confidence level of 60%.

2. Fixed-point inference can make a concern's implicit structure more robust to change.

3. Diversity in intension templates is important to address different concern characteristics.

7.3.3 Threats to Validity

The main threat to the validity of our results is associated with the sample of concern mappings. We used concerns from a previous study, where the task was to compare how different developers would describe the implementation of a high-level concern. The 16 mappings used may not be representative of concern mappings in general for two main reasons. First, they may be unique to the systems or features selected. Second, they were produced by people and as such are subjected to the usual human factors (ability, experience, skill and motivation of the subjects). However, given our intent to model code identified by developers, we believe that our use of human-generated concern models is appropriate. The use of a relatively high number of sample concern mappings (given the cost of obtaining such samples) allows us to decrease the importance of any specific characteristic of a given sample, and thus to achieve a reasonable level of external validity.

7.4 Evaluation: Implementation Tracking

Using historical development data, we simulated how ISIS4J would have performed in a given situation to evaluate its effectiveness, and to gather insights into the kinds of evolutionary changes that could and could not be tracked with our approach. To do so, we studied 34 major releases of the jEdit editor described in Section 7.2. We studied the evolution of jEdit from version 4.0-pre1 (59 KLOC) to version 4.3-pre3 (92 KLOC). We selected the SYNTAX HIGHLIGHTING feature as the main concern of interest because it existed throughout all the studied releases and underwent several changes. Indeed, jEdit supports syntax highlighting of more than 130 file types, and thus must be easily extensible.

This case study enabled us to answer the following research questions:

1. What changes to the implementation of the feature were tracked because of the inferred intensions?

2. What changes could not be captured?

3. Does the intensional representation of concern mappings enable us to better track concerns than their extensional representation?

7.4.1 Experimental Design

We manually created a concern mapping of the SYNTAX HIGHLIGHTING feature in the first and last releases of jEdit we considered. We explored the source code of jEdit using Eclipse, and identified fields and methods that we thought were the most relevant if this feature had to be modified. We limited ourselves to 30 relevant elements, expecting that ISIS4J would then expand this set of interest. In the first and last versions, we identified respectively 24 and 27 elements. We used the mapping of the first version to infer the initial intension set. The mapping of the last version enabled us to compare it with the results of using ISIS4J to track the concern mapping evolution.

To perform our case study, we automated the execution of ISIS4J. We selected a confidence threshold of 60% and inferred intensions until we reached a fixed point in the first version. We then used the same set of intensions throughout all other jEdit versions and did not try to infer new intensions in each version (since the intermediate concerns were not created by developers). Furthermore, we configured the experimental framework as follows:

Intension Conservation. As the system evolves from one version to another, program elements such as methods and fields can be added, removed or changed. Thus, it is possible that an intension's confidence decreases to a point where it no longer meets the required threshold. In this case, we disable the intension from the concern definition, but we keep the elements that were previously matched by that intension. For example, there might exist an intension matching all the five methods accessing a field f_1 in program version 1, with all those five methods being part of the concern. If there are ten methods accessing field f_1 in version 2, confidence drops to 50% (five methods are in the concern mapping out of ten existing methods); thus, the intension will no longer be enabled in the inferred intension set, but the original five methods are kept in the concern mapping. In the current implementation of ISIS4J, intensions are only disabled (not removed) and are automatically re-enabled if they reach the threshold again.

Inconsistencies. It is possible that an element (e.g., a method) identified in the concern was deleted or re-factored between two versions of the system, thus making the concern mapping inconsistent. Our automated experimental framework removed those elements from the concern mapping before applying the intensions on a version of the system. Renamed or moved elements are expected to be included in the concern mapping, provided that (1) they are still matched by one of the intension and (2) the intension's confidence still exceeds the required threshold to be enabled in the intension set.

To summarise our experimental procedure, the manual concern mapping for the first version was taken, and an intension set describing the concern was inferred. Then, those intensions were applied and missing elements in the concern mapping were added. This process was repeated until a fixed point was reached. The modified concern mapping along with the final intension set was then moved to the next version of the system. Inconsistencies in the concern mapping, i.e., program elements that no longer existed in the source code of the new version, were deleted and intensions that no longer reached the threshold were disabled as described above. Intensions that were still enabled were applied to add potentially new elements. This was repeated until we reached the last version where the final tracked concern mapping was compared with the final manual concern mapping.

#	jEdit version name	#	jEdit version name	#	jEdit version name
1	jEdit 4.0 pre1	13	jEdit 4.1 pre3	25	jEdit 4.2 pre7
2	jEdit 4.0 pre2	14	jEdit 4.1 pre6	26	jEdit 4.2 pre10
3	jEdit 4.0 pre3	15	jEdit 4.1 pre8	27	jEdit 4.2 pre11
4	jEdit 4.0 pre4	16	jEdit 4.1 pre9	28	jEdit 4.2 pre13
5	jEdit 4.0 pre5	17	jEdit 4.1 pre10	29	jEdit 4.2 pre14
6	jEdit 4.0 pre6	18	jEdit 4.1 pre11	30	jEdit 4.2 pre15
7	jEdit 4.0 pre7	19	jEdit 4.1	31	jEdit 4.2 final
8	jEdit 4.0 pre8	20	jEdit 4.2 pre1	32	jEdit 4.3 pre1
9	jEdit 4.0 pre9	21	jEdit 4.2 pre2	33	jEdit 4.3 pre2
10	jEdit 4.0 final	22	jEdit 4.2 pre3	34	jEdit 4.3 pre3
11	jEdit 4.1 pre1	23	jEdit 4.2 pre4		
12	jEdit 4.1 pre2	24	jEdit 4.2 pre6		

Table 7.3: List of full version names of jEdit versions used in evaluation

7.4.2 Results

Figure 7.9 on page 89 shows the progress of the following measurements for the Syntax Highlighting concern for each of the 34 versions of jEdit we considered (see Table 7.3 for a mapping between evaluation version number and full jEdit version name): the number of intensions inferred, the average number of concern elements per intension, the concern size (cardinality), as well as the ratio of elements in the concern that are covered by the inferred intensions. We started with 47 intensions in jEdit 4.0-pre1 and ended with 13 intensions in jEdit 4.3-pre3. We have manual concern mapping sizes of 24 and 27 elements respectively, and an average of 3.66 and 2.62 concern elements per intension's extension respectively. Although the intensions in the first jEdit version cover 97% of the concern, this coverage drops to 52% after 34 versions. While this still covers important parts of the concern, we will see how it fails to discover newly added or changed parts of the concern, and why.

Initial Intension-Based Expansion. After ISIS4J inferred the intensions underlying our manual concern mapping, missing elements matching those intensions were added to the concern: the concern mapping size thus went from 24 elements to 40. All the elements added to the concern mapping were considered by the authors to be relevant to the syntax highlighting feature as they contributed to this functionality. As we had limited the size of the manual concern mapping to a maximum of 30 elements, it was expected that ISIS4J would find other relevant elements. For example, the `Token` class, the main parsing unit of syntax highlighting, and two utility classes, `class GUI Utilities` and `TextUtilities`, were added to the concern description along with a subset of their methods: they had not been selected in the manual mapping as they were preferred to elements accessing and using them. For these particular examples, the initial expansion was essential in tracking the concern, since (1) all of the methods identified by the intension-based expansion stayed relevant through the evolution of the concern and (2) one new relevant method was detected for each of these classes.

Changes in Intensions over Time. The overall evolution in the number of intensions can be seen in Figure 7.9. Although the number of intensions stays relatively stable up to jEdit 4.0-final, it drops down to 12 intensions in the following version (4.1-pre1). From then on, the number of intensions stays constant except for an additional one being re-enabled in version 4.2-pre3.

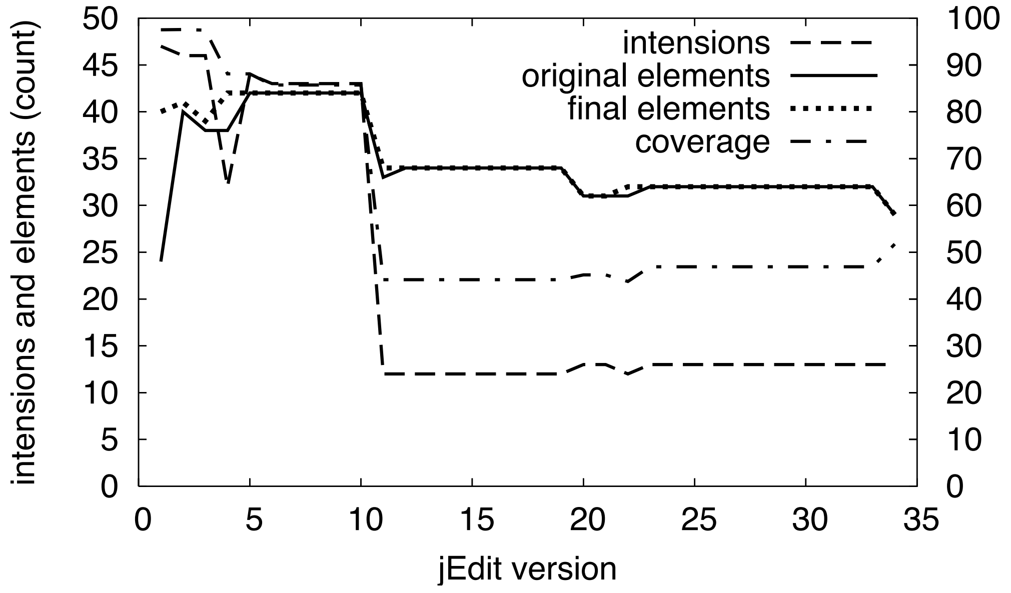
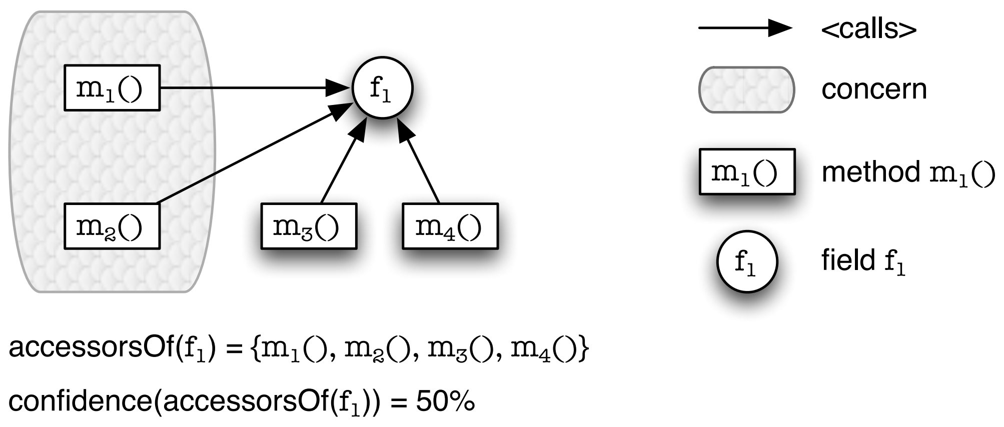


Figure 7.9: Intensions, Concern Elements and Coverage in Each Version of jEdit (see Table 7.3 for reference to actual version names)

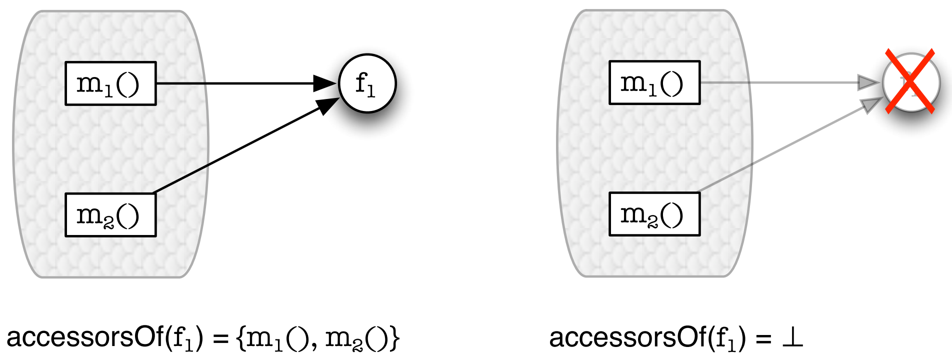
In general, the number of enabled intensions did not vary significantly between two consecutive jEdit versions. There are two main reasons why an intension might be disabled as illustrated by Figure 7.10 on page 90: (a) too many new elements were refactored or introduced in the program between two versions, hence lowering the intension confidence below the selected threshold, or (b) the intension criterion (e.g., field X in accessorsOf(X)) was deleted or renamed/moved. In the evolution of jEdit, 85% of the intensions were disabled for the first reason, and 15% for the second reason.

There was only one major occurrence of intension disabling in the evolution of jEdit, indicated by the drop in the intension count in Figure 7.9 (between jEdit 4.0-final and jEdit 4.1- pre1). Three factors caused this drop. First, the deprecation of one class, `Buffer.TokenList`, removed five elements and disabled three intensions. Second, the move of the method `stringToToken` from class `XModeHandler` to `Token` disabled 14 intensions. This method previously referred to fields only accessed by few other methods. When method `stringToToken` was moved and thus removed from the concern mapping, the intensions' confidence dropped below the threshold. As a result, ISIS4J disabled those intensions, so the move was not captured and subsequent methods that referred to these fields were not included in the concern. The third factor in the major drop in the number of intensions is related to the SYNTAX HIGHLIGHTING feature extension. Since a significant number of new elements were introduced, all of which accessed a given program element, the confidence of intensions whose criterion was that element dropped below threashold, so that they were disabled.

Another interesting irregularity can be observed: a dent in the number of intensions


$m_1()$
f_1
$m_2()$
$m_3()$
$m_4()$
<calls>
concern
method $m_1()$
field f_1
accessorsOf(f_1) = {$m_1()$, $m_2()$, $m_3()$, $m_4()$}
confidence(accessorsOf(f_1)) = 50%

(a) Confidence drops below a given threshold


$m_1()$
f_1
$m_2()$
accessorsOf(f_1) = {$m_1()$, $m_2()$}
$m_1()$
$m_2()$
accessorsOf(f_1) = $\perp$

(b) Program element involved in intension is deleted from code

Figure 7.10: Two reasons why intension `accessorsOf`(f_1) can be disabled

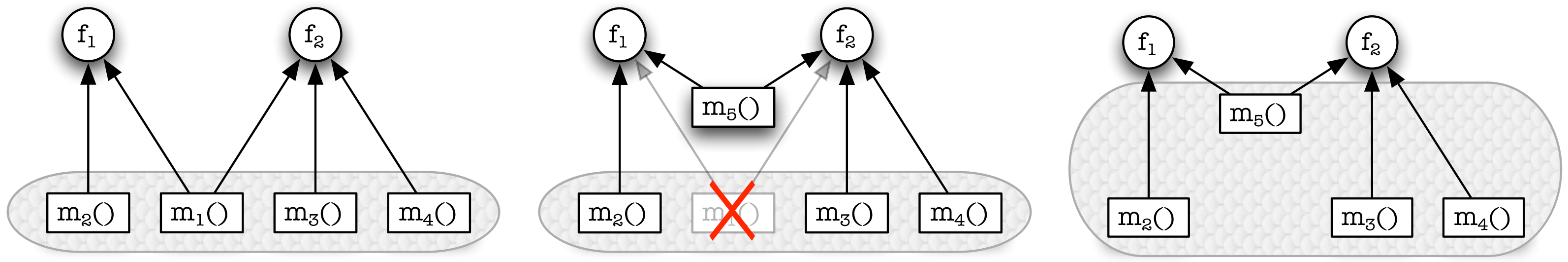
around version 4.0-pre4. Indeed, tracking the SYNTAX HIGHLIGHTING concern in jEdit, the number of intensions drops from 46 to 32 in version 4.0-pre4, and comes back to 44 in version 4.0-pre5. By investigating the code base, we found this was caused by the refactoring of method `loadStyles` that accessed fields used as the criterion in various intensions. Figure 7.11 on page 92 depicts how intensions were temporarily disabled (refactoring of method `loadStyles` is represented by methods m_1 and m_5). In version 4.0-pre4, method m_1 was moved to method m_5 causing some intensions' confidence to drop below the threshold which disabled them (e.g. accessorsOf(f_1)). Since other intensions stayed enabled (e.g. accessorsOf(f_2)), the new method m_5 was added during the intension-based expansion. In the next version, 4.0-pre5, the disabled intensions were reenabled as their confidence reached the threshold again.

Comparison with the Final Manual Concern Mapping. Our approach allowed us to detect four new elements of the SYNTAX HIGHLIGHTING concern throughout the evolution of jEdit. Examples of these elements include the methods `init` in class `Chunk`, or `setStyles` in class `TextArea Painter`. Although none of those four elements was included in our final manual concern mapping, further inspection revealed that they were related to the SYNTAX HIGHLIGHTING feature and should have been included in the concern mapping in the first place. For instance, the `Chunk` class was introduced in version 4.1-pre1 and replaced most of the `Token` class instantiations by extending it. The `setStyles` method on the other hand already existed in the first version of jEdit we studied. However, during its evolution, it became uncohesive and redefined a token type, silently contributing to the SYNTAX HIGHLIGHTING feature. Arguably, those changes were not easy to track manually as they involved only a few lines of code in the existing program.

All new elements in the final manual concern mapping that were not in the original manual mapping remained undiscovered by ISIS4J. For example, the new hierarchy of classes implementing the interface `TokenHandler` was not included by ISIS4J. Further investigation of the code revealed that most of the changes were introduced after version 4.1-pre1 where the massive drop in the intension number occurred. Most of those changes could have been captured by the initial set of intensions provided they had not been disabled because of the refactoring explained previously. Indeed, most new elements that were not captured actually accessed the disabled intensions' criterion.

Finally, there was one element in the ISIS4J final concern mapping that was not in the manual final concern mapping and clearly did not contribute to the SYNTAX HIGHLIGHTING feature: a method in the deprecated class `Buffer.TokenList`. Even though this class was no longer used by any other class in jEdit, and thus was irrelevant, one of the inferred intension in the original concern, declaredBy(`Buffer.TokenList`), ensured that all existing methods of the deprecated class were included in the concern mapping. Only, e.g., additional run-time analysis could remove elements that become irrelevant but still exist. However, we argue this is reasonable as in fact the developers never removed the class from the source code.

Extensional vs. Intensional Concern Definition. We started with 24 concern elements in the first jEdit version. Considering there were 17 inconsistencies that needed to be removed from the concern mapping during the evolution of jEdit, we would have ended up with a concern mapping of only 7 elements if we had used an extensional



accessorsOf(f_1) = {m_1(), m_2()}
accessorsOf(f_2) = {m_1(), m_3(), m_4()}

confidence(accessorsOf(f_1)) = 100%
confidence(accessorsOf(f_2)) = 100%

(a) Intensions before refactoring

accessorsOf(f_1) = {m_1(), m_2()}
accessorsOf(f_2) = {m_1(), m_3(), m_4()}

confidence(accessorsOf(f_1)) = 50%
confidence(accessorsOf(f_2)) = 66.67%

(b) Inconsistencies removed,intension disabled

accessorsOf(f_1) = {m_2(), m_5()}
accessorsOf(f_2) = {m_3(), m_4() , m_5()}

confidence(accessorsOf(f_1)) = 100%
confidence(accessorsOf(f_2)) = 100%

(c) Intension-based expansion

Figure 7.11: Intension wrongly disabled, correctly re-enabled due to fixed point iteration

concern mapping such as provided by ConcernMapper. Using the intentional concern definition that ISIS4J provided, we did better. First, ISIS4J did complete our initial manual concern mapping with 10 further elements, which indicates that completion of concerns using fixed point inference of intentions is helpful. However, one has to bear in mind that it is only helpful if the new elements are indeed appropriate and not false positives. Secondly, ISIS4J removed 17 inconsistencies while an additional 4 relevant elements were found. This gives us 18 concern elements in jEdit's final version rather than 7.

7.4.3 Discussion of Concern Tracking

The use of the jEdit historical development data allowed us to evaluate qualitatively how effective ISIS4J was in tracking changes. The Syntax Highlighting feature went through varied and significant changes through its evolution: 17 elements were removed or refactored and the feature was extended in several ways. Our approach successfully captured feature extensions (e.g. the `Chunk` class), refactorings (e.g. the `loadStyles` method) and small but important changes (e.g. the `setStyles` method) that may have otherwise been difficult to track manually.

We also found that most of our inability to track changes (the small dent in the intension number in version 4.0-pre4, the massive drop of intensions in version 4.1-pre1 and the missed new elements between the manual mappings) were caused by the intensions' confidence level not reaching the 60% threshold. This is an indication that our approach could actually track changes even when major restructurings occur, provided that we lower the threshold.

As we explained in Section 6.2.5, we introduced the threshold variable configured in `Rules.xml` to ensure that we would only infer intensions that were highly representative of a concern's implicit structure even if it did not match it perfectly. ISIS4J also used the same threshold to enable or disable intensions between each version. We wanted to restrict the number of irrelevant elements introduced during the intension-based expansion performed after the initial inference and between each version. However, as it turned out with the Syntax Highlighting feature, the use of a threshold to enable/disable intensions inhibited the tracking of important changes, which suggests that it might be worth lowering it significantly or not using it at all once the initial intensions have been inferred. Indeed, in a manual review of the results, we found that some of the new elements between the original and final manual concern mappings could have been found by the disabled intensions. However, we have to keep in mind that it is still likely to introduce some background noise.

Another important finding is related to the dent in the number of enabled intensions around version 4.0-pre4. Although we ensured that ISIS4J reached a fixed point where intensions could no longer be inferred during the first version, we did not try to reach such a fixed point when doing the intension-based expansion in each subsequent version; we only used it in the first version to address the assumption that the original concern mapping might not have been complete. However, doing fixed point iterations also in subsequent versions would have prevented the temporary drop of intensions. For example, if we had used fixed point inference in Figure 7.11, the $\mathsf{accessorsOf}(f_1)$ intension would have been reenabled in the same version after the addition of m_5 in the mapping instead of waiting for the next version.

Finally, so far we have only investigated seven different intension templates: callerOf, calledBy, accessorOf, accessedBy, declaredBy, implements, and overrides. However, one can think of several other potentially promising templates. For example, if we look at the missed new elements, we see that an intension based on textual similarity would have found those new concern elements while they came up during evolution of jEdit. This observation may simply reflect cut-and-paste, but can also show that developers do give similar names to program elements that do similar things or belong to the same feature. More specifically, if we had an intension that checks for textual similarity, e.g. the same substring in method and field names, we would have identified all but 7 new elements as most of them had at least one of the following substrings in common: ‘keyword’, ‘token’, or ‘rule’.

To summarise, from the evaluation we learnt the following lessons:

1. Intension-based expansion of concern mappings proves to be relevant for concern tracking based on implicit structure.

2. Intensions fail to track changes to a concern mapping if too much refactoring happens from one version to the next, or if the intension criterion gets refactored or deleted.

3. Intensional concern descriptions enable better tracking of concerns than extensional descriptions.

7.4.4 Conclusions

This case study allowed us to better understand how the automatic application of inferred intensions can help track the code associated with a concern, and to gather insights about modification that are unlikely to be tracked. Although it also provides a valuable illustration of the potential benefit of the approach, its external validity is limited in that the case studied cannot be considered representative of the performance of the approach in general. In practice, two main factors will affect the usefulness of the approach: the characteristics of the concern analysed, and the characteristics of the evolution of the code. In order to benefit from our system, a concern mapping must exhibit latent intensions that will be detected and potentially completed by ISIS4J. As seen in Section 7.3, not all concerns have such characteristics. Regarding the impact of the code, the study has helped us demonstrate that the most important benefits will be obtained in conditions of small-scale maintenance, as drastic changes are unlikely to be tractable through existing intensions.

Investigator bias is also a threat to the internal validity of our case study as we both selected the observed feature and produced the original and final concern mappings. Our knowledge of ISIS4J internals could thus have influenced the results of our study. To reduce this threat, we only looked at the first and last version of the subject program to ensure that we would not develop an intimate knowledge with the changes and involuntarily privilege changes that could be tracked by our system. We also ensured that the selected feature went through several different types of structural changes, as can be seen by the high number of inconsistencies and the difference between the original and final mappings. Moreover, one of the purposes of Section 7.3 was to validate experimentally that ISIS4J was able to infer the implicit structure from concern mappings produced by human subjects other than the investigators. Finally, we considered the possibility of using one of the concern mappings used in Section 7.3, but rejected this idea since the concern mappings had been produced on a recent version of the projects, greatly limiting the data points available for an evolution study. It was also not possible to consistently track back the concern mappings to a prior version of the subject programs without introducing a similar bias.

"I never think of the future–it comes soon enough."
Albert Einstein

8 Conclusions and Future Work

Most software change tasks require knowledge about the implementation of different concerns related to the task. This knowledge often has to be manually acquired through reverse-engineering efforts, and is made increasingly harder by the fact that some code is inevitably spread over the code-base in an unstructured manner. This problem gave rise to aspect-oriented programming, which had a surge of popularity in recent years: a cross-cutting concern can be formalised as explicit structure, an aspect. For AOP to be beneficial for existing projects, cross-cutting concerns must be identified first.

Analysing, in particular aspect mining, the development history of a program gives a novel way to identify these aspects. We are the first to use version history to mine aspect candidates. The underlying hypothesis is that cross-cutting concerns emerge and evolve over time. By introducing the dimension of time, and analysing only changes to the system, our aspect mining approach has several advantages. It scales to industrial-sized projects like Eclipse. In particular, we reached top precision (above 90%) for big projects with a long and extensive, detailed history. We have seen that it can discover cross-cutting concerns across platform-specific code (see lock/unlock in Section 4.6). Typically, such concerns are recognised incompletely by static or dynamic approaches which can only investigate one platform at a time, and thus require the mining process to be run multiple times.

Furthermore, we have identified an efficient way to identify all complex aspect candidates efficiently other than by deriving them from mined simple aspect candidates. We proposed to use formal concept analysis for automatic detection. An added benefit of this algebraic theory which identifies conceptual structures among data sets is that its results will include simple aspect candidates since they can be seen as a special form of complex candidate.

Such feature location techniques as introduced here may have to be applied time and again since knowledge about a concern's implementation, in particular documentation, can become invalid as the system evolves. To avoid this and achieve a concern implementation description more robust to evolutionary changes, we suggest to use structural and textual patterns among the program elements for concern tracking. Our approach has several advantages. The inferred patterns are documented as rules that describe a concern in a formal (intensional) rather than a merely textual (extensional) manner. We have seen that these rules support the tracking of a concern's implementation despite, e.g., program code extensions or refactoring activities by developers. Besides, the costs for the automated application of our tracking technique are minimal, e.g., it can be run automatically upon committing changes to the code repository. We can thus help preserve knowledge previously acquired through reverse engineering activites such as aspect mining or tedious code inspections.

Future Work. Besides working on integrating the techniques presented here in tools that can be used by any developer, future work could focus on addressing the following challenges.

It would be interesting to have both approaches, the history-based aspect mining as well as the tracking of evolving concerns via intensions, combined in one tool. New additions to the code would automatically be mined and be the input for automatic updates of concern implementation documentation. An integration of both approaches would also provide additional information that would make the tracking of evolving concerns easier and more precise. For example, if besides additions also deletions of methods are taken into account by the aspect mining side, this can provide information about refactorings that otherwise could get lost.

On the aspect mining side, it would be interesting to see how the concept of localities in the history-based mining approch could be extended when using the formal concept analysis algorithm to identify aspect candidates. One possibility would be to compute the lattice over the sum of commits within a sliding time-window, like the time-window used in our approach for temporal locality.

We have introduced the idea of having rules that describe textual similarity to track features in evolving software. The evaluation has shown that this kind of intension could be useful. Different definitions of textual similarity could be used, from exact word stem matches to the use of synonyms. Particularly textual similarity in the wider sense of synonyms for words (or word stems) will require interdisciplinary research with computer linguistics, but if successful could provide a powerful extension to the current state of the approach.

Bibliography

[1] AspectJ homepage. `http://eclipse.org/aspectj/`.

[2] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM 2005)*, 2005.

[3] Dave Binkley, Mariano Ceccato, Mark Harman, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering (TSE)*, 32:698–717, 2006.

[4] Silvia Breu. Aspect mining using event traces. Master's thesis, University of Passau, Germany, March 2004.

[5] Silvia Breu. Towards hybrid aspect mining: Static extensions to dynamic aspect mining. In *1st Workshop on Aspect Reverse Engineering (WARE) at Working Conference on Reverse Engineering (WCRE)*, November 2004.

[6] Silvia Breu. Extending dynamic aspect mining with static information. In *Proceedings of 5th International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 57–65. IEEE Computer Society, September/October 2005.

[7] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proceedings of 19th International Conference on Automated Software Engineering (ASE)*, pages 310–315. IEEE Press, September 2004.

[8] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In Sebastian Uchitel and Steve Easterbrook, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM Press, September 2006.

[9] Silvia Breu, Thomas Zimmermann, and Christian Lindig. Mining eclipse for crosscutting concerns. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, May 2006.

[10] Magiel Bruntink, Arie van Deursen, Tom Tourwe, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 200–209, Washington, DC, USA, 2004. IEEE Computer Society.

[11] CME website. `http://www.research.ibm.com/cme/`.

[12] CVS homepage. `http://www.nongnu.org/cvs/`.

[13] Barthélémy Dagenais, Silvia Breu, Frédéric Weigand Warr, and Martin P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 254–263, New York, NY, USA, 2007. ACM.

[14] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer, 1995.

[15] Ammon H. Eden and Rick Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering*, pages 149–159, 2003.

[16] Alexander Egyed, Gernot Binder, and Paul Grünbacher. Strada: A tool for scenario-based feature-to-code trace detection and analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07 Companion)*, pages 41–42, 2007.

[17] Alexander Egyed and Paul Grünbacher. Supporting software understanding with automated traceability. *International Journal of Software Engineering and Knowledge Engineering*, 15(5):783–810, 2005.

[18] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, and Audis Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[19] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[20] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[21] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.

[22] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, Berlin, 1999.

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

[24] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[25] William G. Griswold, Y. Kato, and J. J. Yuan. Aspect Browser: Tool support for managing dispersed aspects. Technical Report CS99-0640, UC, San Diego, 1999.

[26] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274, 2001.

[27] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition of legacy code. In *Workshop on Advanced Separation of Concerns*, 2001.

[28] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, 2003.

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

[30] Gregor Kiczales et. al. Aspect-oriented programming. In *Proceedings of 11th European Conf. on Object-Oriented Programming (ECOOP)*, 1997.

[31] Rainer Koschke and Jochen Quante. On dynamic feature location. In *Proceedings of the 20th International Conference on Automated Software Engineering*, pages 420–432, 2005.

[32] Jens Krinke and Silvia Breu. Control-flow-graph-based aspect mining. In *1. Workshop on Aspect Reverse Engineering (WARE) at Working Conference on Reverse Engineering (WCRE)*, November 2004.

[33] Christian Lindig. Fast Concept Analysis. In Gerhard Stumme, editor, *Working with Conceptual Structures – Contributions to ICCS 2000*, pages 152–161, Germany, 2000. Shaker Verlag.

[34] Benjamin Livshits and Thomas Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of European Software Engineering Conference/ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 296–305, New York, NY, USA, 2005. ACM Press.

[35] Neil Loughran and Awais Rashid. Mining aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD Sat. workshop)*, 2002.

[36] Marius Marin, Leon Moonen, and Arie van Deursen. A classification of crosscutting concerns. In *ICSM*, pages 673–676. IEEE Computer Society, 2005.

[37] Marius Marin, Leon Moonen, and Arie van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *13th IEEE Working Conference on Reverse Engineering (WCRE 2006)*, 2006.

[38] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 132–141. IEEE Computer Society, November 2004.

[39] Kim Mens and Andy Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, 2006.

[40] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Fajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 137–148. IEEE Computer Society Press, 2006.

[41] Martin P. Robillard. Tracking concerns in evolving source code: An empirical study. In *22nd IEEE International Conference on Software Maintenance*, pages 479–482, 2006.

[42] Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *24th International Conference on Software Engineering (ICSE)*, pages 406–416, 2002.

[43] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):3, 2007.

[44] Martin P. Robillard and Frederic Weigand-Warr. ConcernMapper: Simple view-based separation of scattered concerns. In *2005 OOPSLA Workshop on Eclipse technology eXchange*, pages 65–69, 2005.

[45] David Shepherd and Lori Pollock. Ophir: A framework for automatic mining and refactoring of aspects. Technical Report 2004-03, U Delaware, 2003.

[46] Elliot Soloway, Robin Lampert, Stanley Letovsky, David Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

[47] Peri Tarr, William Harrison, and Harold Ossher. Pervasive query support in the concern manipulation environment. Technical report, IBM Research, 2005. Report RC23343.

[48] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE-21*, pages 107–119, 1999.

[49] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 112–121. IEEE Computer Society, November 2004.

[50] Tom Tourwé and Kim Mens. Mining aspectual views using formal concept analysis. In *Proc. of Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 97–106. IEEE Computer Society, 2004.

[51] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, June 2005.

[52] Chadd C. Williams and Jeffrey K. Hollingsworth. Recovering system specific rules from software repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 7–11, May 2005.

[53] Tao Xie and Jian Pei. MAPO: Mining API usages from open source repositories. In *Proc. Intl. Workshop on Mining Software Repositories (MSR)*, pages 54–57, May 2006.

[54] Charles Zhang and Hans-Arno Jacobsen. Quantifying aspects in middleware platforms. In *2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 130–139, 2003.

[55] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering*, pages 293–303, 2004.

[56] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. Intl. Workshop on Mining Software Repositories (MSR)*, Edinburgh, Scotland, May 2004.