Mining Eclipse for Cross-Cutting Concerns

Silvia Breu
University of Cambridge
Computer Laboratory
Cambridge, UK
silvia@ieee.org

Thomas Zimmermann
Saarland University
Dept. of Computer Science
Saarbrücken, Germany
tz@acm.org

Christian Lindig
Saarland University
Dept. of Computer Science
Saarbrücken, Germany
lindig@cs.uni-sb.de

ABSTRACT

Software may contain functionality that does not align with its architecture. Such cross-cutting concerns do not 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 apply formal concept analysis—an algebraic theory. History-based mining scales well: we are the first to report aspects mined from an industrial-sized project like Eclipse. For example, we identified a locking concern that crosscuts 1284 methods.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

General Terms

Languages, Documentation, Algorithms

1. INTRODUCTION

As object-oriented programs evolve over time, they may suffer from *"the tyranny of dominant decomposition"* [15]: The program can be modularised in only one way at a time. Concerns that are added later and that no longer align with that modularisation end up scattered across many modules and tangled with one another. Aspect-oriented programming (AOP) remedies this by factoring out aspects and weaving them back in a separate processing step [7]. For existing projects to benefit from AOP, these cross-cutting concerns must be identified first. This task is called *aspect mining*.

MSR'06, May 22–23, 2006, Shanghai, China.

We solve this problem by taking a historical perspective: Our analysis is based on the hypothesis that cross-cutting concerns are added to a project over time. A code change in the history of a program is likely to introduce such a concern if the modification gets introduced to various locations within a single code change. This observation is our conceptual contribution.

Our hypothesis is supported by the following example: On November 10, 2004, Silenio Quarti committed code changes "76595 (new lock)" to the Eclipse CVS repository. These changes fixed the bug #76595 "Hang in gfk_pixbuf_new" that reported a deadlock[1] and required the implementation of a new locking mechanism for several platforms. The extent of Silenio Quarti's modification was immense: He modified 2 573 methods and inserted in 1 284 methods a call to the `lock` method, as well as a call to an `unlock` method. Obviously AOP could have been used to weave in this locking mechanism.

For the locking mechanism of Eclipse, it turns out that the locations where calls to `lock` were inserted are exactly the same as the locations where calls to `unlock` were added. This is why we combine the two simple aspect candidates into a *complex aspect candidate*: `lock`, `unlock` were added in 1 284 different locations. However, in the presence of many complex aspect candidates it is not obvious how to find them efficiently. We propose to use *formal concept analysis* [4] for automatically detecting complex aspect candidates, which is our technical contribution and detailed in the next section.

2. MINING CROSS-CUTTING CONCERNS

Previous approaches to aspect mining considered only a single version of a program using static and dynamic program analysis techniques. We introduce an additional dimension: the *history* of a project. Technically, we mine version archives for aspect candidates.

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 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

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

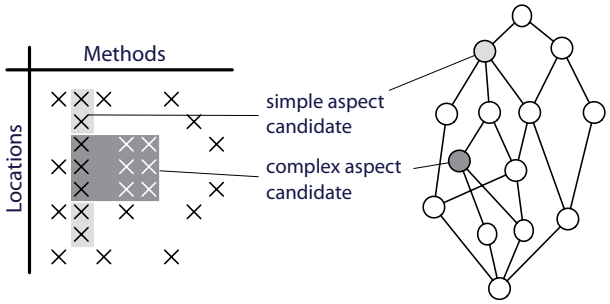

Figure 1: Maximal blocks represent aspect candidates in a transaction (left). Here, 14 candidates form a lattice of super and sub aspects (right). A sub aspect (dark) crosscuts fewer locations but calls more methods than a super aspect (light).

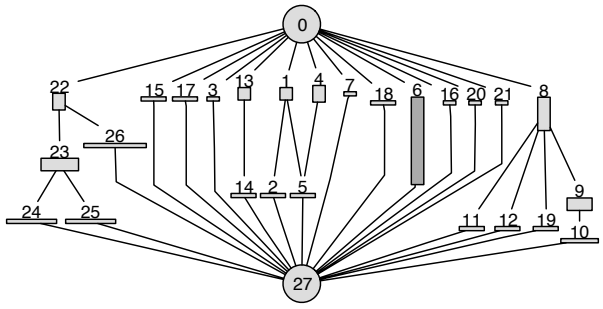

Figure 2: The lattice of aspect candidates from a commit to Eclipse CVS on 2004-03-01 by developer `ptff`. Candidate 6 contains 14 additions of calls to `unsupportedIn2()`.

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.

Aspects are maximal Blocks. We can think of a transaction as a cross table with locations as rows and methods as columns (Figure 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, a candidate is a pair (L, M) of locations L and methods M with $|M| = 1$ and $|L| \geq 7$ for simple candidates.

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 1 such a block is marked by the grey-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 dark-grey-shaded rectangle of size 3×3 in Figure 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 [4]. A maximal block is a pair (L, M) where the following holds:

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

Each block (L, M) is maximal in the following sense: we can't add another method m to M without shrinking L to ensure that *all* locations in L call m. Likewise, we can't add another location l to L without shrinking M. The definition allows for blocks of any size. However, we only consider blocks with $|L| \geq 7$ as aspect candidates. 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 [9]. This is not a concern here since we compute the blocks for each transaction individually. 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 (c.f. Figure 1, right). In our experience, aspects in one transaction are rarely in a super/sub order but typically unordered.

3. EXAMPLES

Figure 2 shows the lattice of all aspect candidates from an Eclipse CVS 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*,

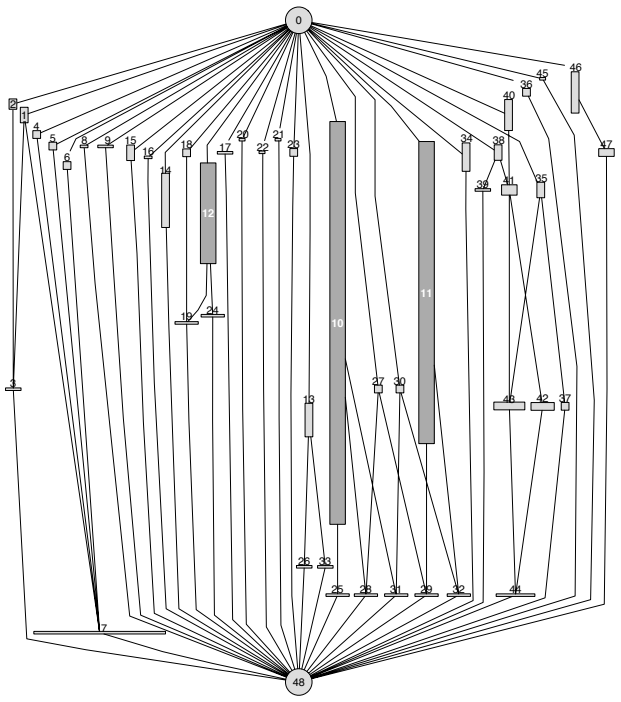

Figure 3: The lattice of aspect candidates from a commit to Eclipse CVS on 2004-02-25 by developer `ptff`. Candidate 10, e.g., contains 104 additions of calls to `preReplaceChild(3)`, `postReplaceChild(3)`.

expression, *operator*, or *body* that are associated with a node in an abstract syntax tree.
This is a typical example for a *role super-imposition* concern [12]. 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 3.

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 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.

4. DATA COLLECTION

Our mining approach can be applied to any version control system, however, we based our implementation on CVS since most open-source projects are using 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 [20] with a 200 seconds window. A reconstructed commit consists of a set of revisions R where each revision $r \in R$ is the results 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 $r \in R$ and its predecessor and compute the set of all calls C_1 in r and C_0 for the preprocessor by traversing the ASTs. Then $C_r = C_1 \setminus C_0$ is the set of inserted calls within r; the union of all C_r for $r \in R$ forms a *transaction* $T = \bigcup_{r \in R} C_r$ which serves as input for our aspect mining.

Unlike Williams and Hollingsworth [18, 19], our approach does not build (compile, link) *snapshots* of a system to compute inserted method calls. As they point out, such interactions with the build environment (compilers, make files) are extremely difficult to handle and result in high computational costs. Instead, we analyse only the differences between single revisions. As a result, our preprocessing is cheap, as well as platform- and compiler-independent; the drawback is that types cannot be resolved because only one file is investigated. In particular, we miss the signature of called methods. In order to reduce name collision, we use the number of arguments in addition to method names to identify methods calls. We believe this is good enough because we are analysing one transaction at a time.

5. RELATED WORK

While this work is not the first that applies formal concept analysis as static analysis to mine cross-cutting functionality, it is the first that leverages software repositories to do so. Furthermore, our approach is the first that scales to industrial-sized projects such as Eclipse.

Static Aspect Mining. The Aspect Browser [5] identifies cross-cutting concerns with textual-pattern matching (much like "grep") and highlights them. The Aspect Mining Tool (AMT) [6] combines text- and type-based analysis of source code to reduce false positives. Ophir [14] uses a control-based comparison, applying code clone detection on program dependence graphs. Tourwé and Mens [17] introduce an identifier analysis, that is based on formal concept analysis for mining aspectual views such as structurally related classes and methods. Krinke and Breu [8] propose an automatic static aspect mining based on control flow. The control flow graph of a program is mined for recurring execution patterns of methods. The fan-in analysis by Marin, van Deursen, and Moonen [13] determines methods that are called from many different places—thus having a high fan-in. Our approach presented here is similar to the fan-in analysis. However, with access to serveral versions of a program we can rule out certain such functions as non cross-cutting and therefore are more precise.

Dynamic Aspect Mining. DynAMiT (Dynamic Aspect Mining Tool) [1, 3] is a dynamic approach that analyses program traces reflecting the run-time behaviour of a system in search for recurring execution patterns of method relations. Tonella and Ceccato [16] suggest a technique that applies concept analysis to the relationship between execution traces and executed computational units (methods).

Hybrid Techniques. Loughran and Rashid [11] investigated possible representations of aspects found in a legacy system in order to provide best tool support for aspect mining. Breu also reports on a hybrid approach [2] where the

dynamic information of the previous DynAMiT approach is complemented with static type information such as static object types.

Mining Co-change. 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, are related to each other.* 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.

Mining Co-addition of Method Calls. 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 used this observation to mine pairs of functions that form usage patterns from version archives [19]. Livshits and Zimmermann used data mining to locate patterns of arbitrary size and applied dynamic analysis to validate their patterns and identify violations [10]. 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.

6. CONCLUSIONS

We are the first who leverage version history to mine aspect candidates. Previous approaches considered a program only at a particular time, using traditional static and dynamic program analysis techniques. One fundamental problem is their *scalability*. In contrast, our history-based aspect mining approach scales well to industrial-sized projects such as Eclipse with million lines of codes.

Formal concept analysis provides a framework to mine and understand aspect candidates: A transaction is a relation over locations and methods where aspect candidates are the maximal blocks of this relation. These form a lattice of super and sub concepts and can be computed efficiently.

Besides general issues such as performance or ease of use, our future work will concentrate on the following topics:

Measure precision We plan to evaluate our technique by manually investigating the top-ranked aspect candidates to check whether they are actual cross-cutting concerns. The resulting precision will measure the effectiveness of our approach.

Combine several transactions Cross-cutting concerns are frequently introduced within one transaction and extended to new locations in later transactions. Although such concerns are recognised by our technique as several aspect candidates, these candidates may be missed. To locate such aspect candidates, we will use *localities*. For instance, two transactions are related if they changed the same locations or were created by the same developer.

For future and related work regarding history-based aspect mining, see:

`http://www.st.cs.uni-sb.de/softevo/`

7. REFERENCES

[1] S. Breu. Aspect Mining Using Event Traces. Master's thesis, University of Passau, Germany, Mar. 2004.

[2] S. 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, Sept./Oct. 2005.

[3] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *Proceedings of 19th International Conference on Automated Software Engineering (ASE)*, pages 310–315. IEEE Press, Sept. 2004.

[4] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, Berlin, 1999.

[5] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. Technical Report CS99-0640, UC, San Diego, 1999.

[6] J. Hannemann and G. Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns*, 2001.

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

[8] J. Krinke and S. Breu. Control-Flow-Graph-Based Aspect Mining. In *1. Workshop on Aspect Reverse Engineering (WARE) at Working Conference on Reverse Engineering (WCRE)*, Nov. 2004.

[9] C. Lindig. Fast concept analysis. In G. Stumme, editor, *Working with Conceptual Structures – Contributions to ICCS 2000*, pages 152–161, Germany, 2000. Shaker Verlag.

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

[11] N. Loughran and A. Rashid. Mining Aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD)*, 2002.

[12] M. Marin, L. Moonen, and A. van Deursen. A classification of crosscutting concerns. In *ICSM*, pages 673–676. IEEE Computer Society, 2005.

[13] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 132–141. IEEE Computer Society, Nov. 2004.

[14] D. Shepherd and L. Pollock. Ophir: A Framework for Automatic Mining and Refactoring of Aspects. Technical Report 2004-03, U Delaware, 2003.

[15] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE-21*, pages 107–119, 1999.

[16] P. Tonella and M. 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, Nov. 2004.

[17] T. Tourwé and K. 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.

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

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

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