HAM: Cross-cutting Concerns in Eclipse

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

As programs evolve, newly added functionality sometimes does no longer align with the original design, ending up scattered across the software system. Aspect mining tries to identify such cross-cutting concerns in a program to support maintenance, or as a first step towards an aspect-oriented program. Previous approaches to aspect mining applied static or dynamic program analysis techniques to a single version of a system. We leverage all versions from a system's CVS history to mine aspect candidates with our Eclipse plug-in HAM: when a single CVS commit adds calls to the same (small) set of methods in many unrelated locations, these method calls are likely to be cross-cutting. HAM employs formal concept analysis to identify aspect candidates. Analysing one commit at a time makes the approach scale to industrial-sized programs. In an evaluation we mined cross-cutting concerns from Eclipse 3.2M3 and found that up to 90% of the top-10 aspect candidates are truly cross-cutting concerns.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and re-engineering

General Terms Algorithms, Measurement, Documentation, Performance, Design, Experimentation,

Keywords Aspect Mining, Aspect-Oriented Programming, CVS, Eclipse, Formal Concept Analysis, Java, Mining Version Archives

1. Introduction

As a program evolves it is easy to overlook that certain functionality is not or no longer properly encapsulated but scattered over many methods. Aspect mining aims at identifying such cross-cutting concerns, also referred to as *aspects*. Aspects constitute structural problems that either have to be takes care of manually, through object-oriented refactoring, or by moving towards aspect-oriented programming (AOP). However, we believe that aspects do not necessarily exist from the beginning but may be introduced over time to a system. Motivated by dynamic approaches for aspect mining that investigate execution traces of programs [2, 3], we build our analysis on CVS commits that insert method calls. We are working on an Eclipse plug-in called HAM that will identify such cross-cutting concerns and will inform the programmer unobtrusively when she is about to add more such functionality. She might then go on as planned, or think about introducing an abstraction to encapsulate this functionality properly. HAM employs formal concept analysis to compute all potential aspects from which we filter the most likely ones. Aspects from a CVS commit may not be independent but form a hierarchy. Besides showing all instances of an aspect candidate, HAM also visualises this hierarchy to inform the user about potentially conflicting aspects.

2. Examples from Eclipse

In Eclipse, we found numerous aspect candidates, of which a few are presented in more detail in the following.

Locking Mechanism. 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.

Bytecode Visitor. Another example for a cross-cutting concern is the call to method `dumpPcNumber` which was inserted to 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 method 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();
}
```

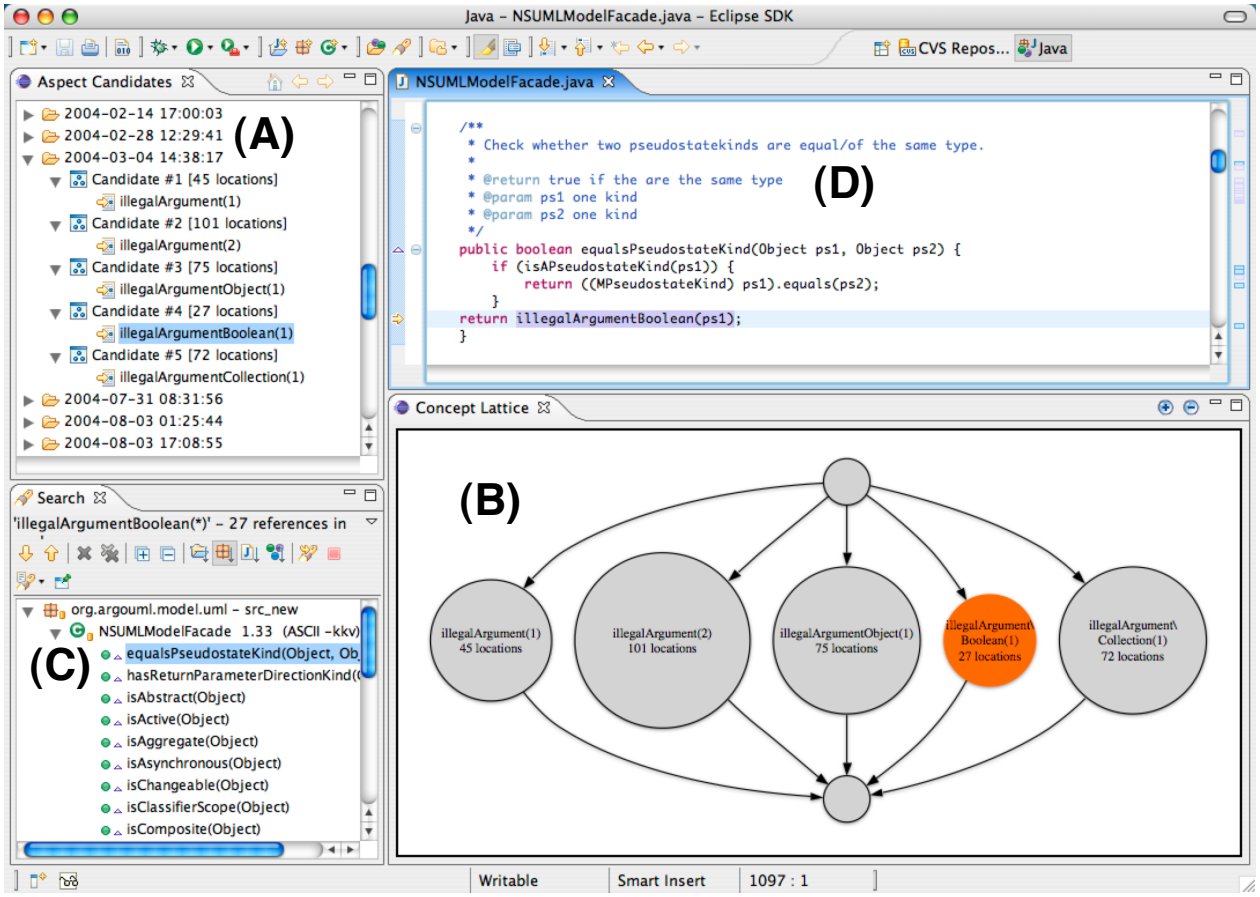

Figure 1. The HAM plug-in. The *Aspect Candidates* view (A) lists cross-cutting concerns that can be investigated with other views (B,C,D). View (B) shows the hierarchy of aspect candidates.

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

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

This is a typical example of a *role super-imposition* concern [8]. 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 `postValueChange` 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;
}
```

3. The HAM Plug-in

So far we have implemented a prototype of HAM that identifies cross-cutting concerns from CVS archives and presents the results in Eclipse. In our future work, we will inform the programmer unobtrusively when she is about to change cross-cutting functionality.

Figure 1 shows a screenshot when analysing ArgoUML [1] (a UML modelling tool) for cross-cutting concerns. In the left pane **(A)**, the view *"Aspect Candidates"* lists all transactions

of the CVS repository for which we found aspect candidates. For the transaction on April 4 2004 HAM finds five candidates: `illegalArgument(1)` (for which a call was inserted into 45 locations), `illegalArgument(2)` (101 locations), `illegalArgumentObject(1)` (75 locations), `illegalArgumentBoolean(1)` (27 locations), and `illegalArgumentCollection(1)` (72 locations).

Double clicking a transaction opens the corresponding lattice in view *"Concept Lattice"* **(B)** on the lower right hand side. This view allows to explore the relationship between candidates. The middle layer of five nodes represent the five aspect candidates that we found in this particular transaction. In this case, there is no path from the top to the bottom node that visits two aspect candidates. Thus, the locations of the candidates are disjoint and unlikely to interfere.

Double clicking an aspect candidate (in any view) opens the *"Search"* view **(C)** of Eclipse in the lower left pane. This view lists all locations where a candidate was inserted. In our example, `illegalArgumentBoolean` was called in 27 location—among them `equalsPseudostateKind`. We can now inspect the code in the editor on the upper right hand side **(D)** and verify that the candidate is actually cross-cutting.

4. Underlying Technique

In the following, we describe in more detail what our prototype HAM and approach builds upon.

4.1 Preprocessing

Our approach can be applied to any version control system. However, we based our implementation on CVS since most open source projects currently use it. First, we reconstruct CVS commits with a *sliding time window* approach [10]. A reconstructed commit consists of a set R of revisions where each revision $r \in R$ is the result of a single check-in.

Additionally, we compute method calls that have been inserted within a commit operation R. A commit R is a set of changed locations—in our case locations are method bodies but could be classes or packages as well. For every location l that was changed in R we compute the set $M(l)$ of added method calls by comparing the abstract syntax tree of l before and after commit R. As a result we obtain a set $T(R) = \{(l, m) \mid l \in R, m \in M(l)\}$ of *new calls* from location l to method m. We call a set $T(R)$ of new calls a *transaction*; transactions serve as main input for our aspect mining. Here is an example from the Eclipse project:

$$\left\{\begin{array}{l}(\texttt{DefaultBytecodeVisitor.\_aaload(int)}, \texttt{dumpPcNumber(1)}),\\ (\texttt{DefaultByteCodeVisitor.\_aastore(int)}, \texttt{dumpPcNumber(1)}),\\ (\texttt{DefaultByteCodeVisitor.\_aload(int, int)}, \texttt{dumpPcNumber(1)})\end{array}\right\}$$

Into three locations `_aaload`, `_aastore`, and `_aload` a call to method `dumpPcNumber(1)` was inserted. In order to reduce computational cost, we analyse only the differences between single revisions but not between the resulting programs before and after a revision. Therefore we cannot resolve signatures for called methods. Instead we use their names (e.g., `dumpPcNumber`) and number of arguments (e.g., 1). For more details on our preprocessing, we refer to the APFEL plug-in [9].

4.2 Mining Transactions

For our analysis, history (of a program) is a sequence of transactions. Each transaction is a set of added method calls (l, m) from location l to method m[1]. A transaction T is formally a relation and can be depicted as a cross table between locations L and methods M—cf. Figure 2 .

[1] We ignore changes and deletions of calls as we are only interested in aspects emerging over time.

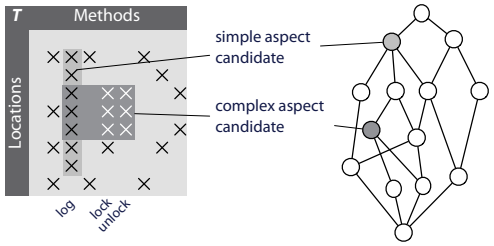

Figure 2. A transaction $T \subseteq \mathcal{L} \times \mathcal{M}$ is a relation between locations $\mathcal{L}$ and methods $\mathcal{M}$. The maximal (rectangular) blocks of T are aspect candidates, which form a hierarchy.

Simple Aspects. When a transaction inserts calls to a logging method `log` in 10 locations these calls show up in the cross table as a *block* of size 1×10 (given an appropriate order of locations). We consider adding a call to be an aspect candidate when it cross-cuts at least 8 locations. At each location where a call to `log` was added, calls to other methods may have been added as well. Still, aspects where a call to a single method (like `log`) is added are simple to detect in a transaction by sorting calls (l, m) by the called method m. We call these *simple aspect candidates*. Obviously a candidate is more likely to be a genuine aspect when the number of locations it cross-cuts is high.

Complex Aspects. Some aspects come as pairs of function calls: a call to `lock` for locking a resource is typically followed by a call to `unlock`. Given an appropriate order of rows *and* columns, the addition of calls to `lock` and `unlock` in 10 locations also shows up as a (2×10) block in the cross table. We call the addition of calls to two or more methods a *complex aspect* candidate. Again, we consider such a block only a candidate if it cross-cuts at least 8 locations. Unlike simple aspect candidates, it is not obvious how to detect such complex aspect candidates in a transaction efficiently.

4.3 Formal Concept Analysis

The problem of identifying all blocks is the subject of formal concept analysis, an algebraic theory for binary relations [6], which also provides efficient algorithms [7]. A maximal block in a transaction $T \subseteq \mathcal{L} \times \mathcal{M}$ is a pair (L, M) of locations and methods where the following holds:

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

Formal concept analysis considers all blocks in a relation, not just those exceeding certain limits. The definition of blocks in particular allows for blocks with one empty component and subsumes simple and complex aspect candidates. We therefore compute all blocks and filter them later for aspect candidates.

Interestingly, blocks and therefore aspects form a lattice, defined by the partial order $(L, M) \leq (L', M') \Leftrightarrow L \leq L'$. However, typically the aspect candidates of a transaction are incomparable. Figure 3 shows the lattice of blocks for such a transaction from the Eclipse project.

5. Experience and Results

Because a cross table of size $n \times n$ may have up to 2^n blocks, concept analysis is potentially expensive. This has not been a problem so far: for 43 270 transactions in the Eclipse CVS repository, the average transaction adds 5.4 calls in 3.8 locations and has 3.7 blocks. However, the largest transaction had 1235 blocks. On average, computing all blocks for a transaction took less than 1 sec.

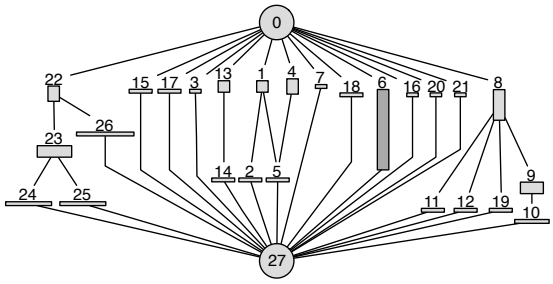

Figure 3. Hierarchy of blocks from an Eclipse transaction. Block 6 is an aspect candidate, cross-cutting 14 locations.

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

Table 1. Aspect candidates mined from 43 270 CVS transactions for Eclipse 3.2M3. There are 88 candidates that added exactly 3 method calls.

The 43 270 transactions of the Eclipse CVS archive constitute 159 448 blocks. From these we mined 2353 aspect candidates, with the distribution shown in Table 1. We found 1878 simple and $363 + 88 + 24 = 475$ complex candidates.

In [4] we had previously mined Eclipse for simple and complex aspect candidates, albeit with a less general approach. There we reported 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). With our new approach we found 64 unique aspect candidates, including all 31 aspect candidates reported in [4]. This confirms our two claims: formal concept analysis provides the right formal and algorithmic framework to mine aspects, and aspects can be mined efficiently from large projects by analysing code additions over time.

6. Contributions

We are the first to leverage version history to mine aspect candidates. The underlying hypothesis and motivation is that cross-cutting concerns may emerge over time. Our work shows that version archives are indeed useful for aspect mining.

HAM adds a new dimension to aspect mining. Previous work on aspect mining considered only a particular version of a program. Our approach uses project history as additional input. This enables a new view on the evolution of aspects.

HAM scales and is platform independent. HAM is the first aspect mining approach that scales to industrial-sized projects like Eclipse. Furthermore, it recognises cross-cutting concerns across code for different platforms.

HAM comes with high precision. We thoroughly evaluated 405 aspect candidates returned by HAM [4]. The precision increases with project size and history, for Eclipse up to 90% for the top-10 candidates. For small projects, HAM suffers from the much fewer data available, resulting in lower precision (about 60%).

For for more information on HAM, log on to:

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

References

[1] ArgoUML. ArgoUML project homepage. http://argouml.tigris.org/.

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

[3] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *Proc. of 19th Intl. Conf. on Automated Software Engineering (ASE)*, pp. 310–315. IEEE Press, 2004.

[4] S. Breu and T. Zimmermann. Mining Aspects from Version History. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Tokyo, Japan, September 2006.

[5] S. Breu and T. Zimmermann and C. Lindig. Mining Eclipse for Cross-Cutting Concerns. In *Proc. Intl. Workshop on Mining Software Repositories (MSR)*, Shanghai, China, May 2006.

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

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

[8] M. Marin, L. Moonen, and A. van Deursen. A Classification of Crosscutting Concerns. In *ICSM*, pp. 673–676. IEEE Computer Society, 2005.

[9] T. Zimmermann. Fine-grained Processing of CVS Archives with APFEL. Technical Report, Saarland University, Saarbrücken, Germany, 2006. Available online at: http://www.st.cs.uni-sb.de/softevo/

[10] 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.