Identifying Cross-Cutting Concerns from History

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

Thomas Zimmermann
Saarland University, Germany
tz@acm.org

1 Motivation

As object-oriented programs evolve, they may suffer from the "tyranny of dominant decomposition": The program can be modularised only one way at a time, leaving cross-cutting concerns scattered across many modules and tangled with one another. Aspect-oriented programming (AOP) tries to remedy this by encapsulating these concerns into aspects. *Aspect mining* identifies such cross-cutting concerns and thus helps to migrate existing software to an aspect-oriented program.

Aspect mining of large software system like Eclipse is a problem: dynamic approaches depend on test cases and have trouble covering all code and thus detecting all cross-cutting functionality. And static approaches simply cannot analyse systems this big unless they work incrementally.

We offer a new approach based on the observation that cross-cutting functionality does not exist from the beginning. Instead, it is introduced over time. More specifically, we speculate that considerable cross-cutting functionality is introduced within short periods of time. To find these, we analyse code additions from development tasks as they are recorded in a software repository like CVS. Since we analyse one task at a time, our approach is independent of a project's total size. This enables us to report cross-cutting functionality for Eclipse, a 1.6 MLOC Java program. In this paper we sketch the basic idea of history-based aspect mining and some initial results.

2 History-based Aspect Mining

Previous approaches to aspect mining considered only a single version of a program using static and dynamic program analysis techniques. We introduce the additional dimension of time by mining CVS *transactions* that introduce new code. Within each transaction we identify those changes that are likely to introduce cross-cutting concerns, which we call *aspect candidates*.

2.1 Transactions and Aspect Candidates

The history of a program can be modelled as a sequence of transactions. A single transaction collects all code changes between two program versions made by a programmer to complete one development task. From a technical point of view, a transactions is defined by the kind of version archive we analyse. We analyse projects managed with CVS, but our approach extends to any version archive.

Motivated by our previous dynamic aspect mining approaches that analysed program traces [1, 2], we base our history-based technique on changes that insert or delete method calls. As we are interested in the *introduction* of cross-cutting concerns, we take a simplified view and concentrate on method call additions only, omitting deletions. This leads to the following model of a transaction: A transaction is a set of additions, each represented by a pair (l, m) of a location and a method. The location l denotes where the call was added (a method body), and the method m denotes the call that was added.

The addition of a method call m (like `notify()`) in a location l is likely to be cross-cutting if the *same* call (to `notify()`) is added in many other locations as well. To avoid false positives, we require an aspect candidate to cross-cut at least eight locations. Hence, we partition a transaction into sets of additions that introduce a call to the same method but only consider sets with at least 8 elements.

2.2 Ranking Aspect Candidates

Our goal is not an automatic refactoring based on our analysis. Instead, we like to rank aspect candidates for manual inspection. Our first criterion is to rank aspect candidates by the number cross-cut locations. However, this still leads to many false positives: calls to methods like `hasNext()` are frequently inserted into many locations but do not represent cross-cutting functionality.

Because calls to methods like `hasNext()` or `print()` are added frequently, these additions are present in many transactions. On the other hand, calls to truly cross-cutting functionality are much less frequent. We therefore consider the *fragmentation* of an aspect candidate: the number of transactions where it was found. A candidate with low fragmentation is ranked higher than a candidate with high fragmentation; candidates of the same fragmentation are ranked by size.

Manual inspection of our ranking revealed that some true aspect candidates ranked low because they were introduced in one transaction and extended later, possibly by a different developer. We therefore chose to abandon ranking by fragmentation in favour of

ranking by *compactness*: the ratio between the cross-cut locations in the transaction at hand and the total number of locations where that call was added in other transactions. We thus calculate compactness per transaction and per candidate; for our ranking we only consider the maximal compactness of a candidate. Ranking by compactness ranks common calls like `hasNext()` low but does not penalise cross-cutting calls that are spread over few transactions.

3 First Experiences

We performed an initial evaluation of history-based aspect mining by applying it to Eclipse 3.2M3. It currently comprises 1 675 kLOC in just under 13 000 classes with more than 74 000 methods. We chose this industrial-sized project for it's many developers and large history. First results are very promising and support our hypothesis that cross-cutting may not exist from the beginning but emerge over time. The following example illustrates that.

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 (see `https://bugs.eclipse.org/`) and required the implementation of a new locking mechanism for several platforms. The extent of the 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.

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 code below shows the method for instruction `aload_0`.

```
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 can even generate all 205 methods (including comment) since the methods differ only in the name of the bytecode instruction.

4 Related Work

We are the first to introduce the additional dimension of time to aspect mining by leveraging version history to mine aspect candidates. Previous approaches were either dynamic, static, or hybrid program analysis techniques, none of which dealt with several versions of a software system. We believe that with our work we will not only be scalable and have high precision. We can also overcome the usual problems of too low test case coverage or false positives due to analysing non-executable code.

However, mining software repositories itself has been used before, such as mining co-change, and more recently the extension to mining co-addition of method calls. Williams and Hollingsworth use it to mine pairs of functions that form usage patterns from version archives [4]. Livshits and Zimmermann use it to locate patterns of arbitrary size and apply dynamic analysis to validate their patterns and identify violations [3]. Our approach also considers the addition of method calls, however, we do not focus on calls that are inserted together but on locations where the same call is inserted. Thus, we identify cross-cutting concerns rather than usage patterns.

5 Conclusions and Consequences

Analysing a project's history to identify cross-cutting concerns proved to be promising, scaling well to industrial-sized projects with millions LOC. Thus, we are confident that taking lessons from history will help us to improve and learn about today's software.

We plan a thorough evaluation. We aim at mining other open-source projects as well as the aspect mining benchmark software system JHotDraw. This includes a careful evaluation of all results from these projects as well as of the tremendous amount of aspect candidates mined from Eclipse, including the calculation of our approach's precision, and a comparison of how the different ranking techniques perform. Furthermore, we plan to cross-validate our approach with other aspect mining approaches as well as to compare its performance with other version archive analysis techniques.

References

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

[2] S. Breu. Extending Dynamic Aspect Mining with Static Information. In *Proc. 5th International Workshop on Source Code Analysis and Manipulation (SCAM)*, pp. 57–65. IEEE Computer Society, Oct. 2005.

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

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