Aspect Mining for Aspect Refactoring: An Experience Report

Maximilian Störzer, Uli Eibauer and Stefan Schoeffmann
Universität Passau, Passau, Germany
{stoerzer, eibauer}@fmi.uni-passau.de, stefan.schoeffmann@sdm.de

ABSTRACT

Aspect-Oriented programming currently suffers from one increasingly important problem–while there is an abundance of aspect-oriented languages and systems, only few example programs are publicly available. To lighten this situation, we set out to refactor crosscutting concerns into aspects for Open Source Java systems.

Aspect Mining (AM) is an important enabler of Aspect-Oriented Refactoring (AOR), and this paper reports about our preliminary experience with automatic and manual aspect refactoring. From this experience we formulate interesting research questions for further research.

1. MOTIVATION

Aspect-Oriented Programming has been proposed to address limitations in current programming paradigms called the *tyranny of the dominant decomposition* in literature [8]. While there is an abundance of available languages, currently only few non-trivial examples programs are publicly available.

To address this lack of code we started two projects related to aspect-oriented refactoring in the spirit of related work on (A)JHotDraw [9, 1]. Our projects were originally independent of each other: first, we designed an automatic refactoring tool[7] for Eclipse based on fully automatic aspect mining[1] using DynAMiT[3] and conducted three case studies with this tool. Second we are currently working on a project targeted to refactor the open source Java application HSQLDB (http://hsqldb.org/) using a semantics-guided approach. Both our tool and the refactoring project use AspectJ as target language.

The purpose of our refactoring tool was to easily generate AspectJ programs out of Open Source Java applications. However, this goal turned out to be very ambitious. Nevertheless we learned some important lessons for usability of aspect mining results for automatic refactoring which we report in this paper.

HSQLDB is a medium-size open source project (65 kLoC), implementing an relational database system. HSQLDB comes with a JUnit test suite which we use to guarantee functional equivalence of our system. Clearly the first step for an aspect-oriented refactoring is to find relevant crosscutting concerns which actually can be refactored as aspects. We used manual semantics-guided code inspection supported by FEAT[6] to find relevant crosscutting code.

As we originally did not intend to use these two projects to evaluate aspect mining techniques, we did not perform our case studies using the same projects. However, we argue that the basic observations and results we report here are inherent to the underlying techniques—automatic versus manual aspect refactoring—and are thus an interesting contribution, even if we agree that a more thorough case study on one subject to verify these results is needed.

The contributions of this paper are twofold. First we report our experience from two aspect-refactoring projects–one conducted with automatic, one with manual aspect mining–and derive interesting research questions for aspect mining tools from a comparison of this experience. Second, as for HSQLDB both the Java and the AspectJ version will be available once our project is finished, our effort will also result in an interesting evaluation test case for (new) aspect mining tools. Comparing results of AM tools to the aspects we found by manually analyzing the system might be an interesting benchmark.

2. AUTOMATIC AOR WITH DYNAMIT

DynAMiT is an automatic aspect mining tool based on dynamic program analysis. The tool evaluates traced call sequences to discover repeated patterns, which are then–if certain thresholds in repetition are reached–reported as aspect candidates.

DynAMiT discovers candidates for `before` and `after`-advice for `call` and `execution` joinpoints. For example DynAMiT discovers that each time `f()` is called, `g()` is called immediately after and then suggests that the call to `g()` should be embedded in an `after`-advice to the call to `f()` or, symmetrically, the call to `f()` should be embedded in a `before`-advice to the call to `g()`.

Our automatic aspect refactoring tool uses DynAMiT to find aspect candidates, analyzes its results to figure out if a refactoring is feasible, and, if so, allows to automatically refactor aspect candidates. For our analysis we check if candidates identified by DynAMiT can be moved to an aspect (using AspectJ) without changing program semantics. Therefore, context at the joinpoint has to be available for AspectJ, and values (after the joinpoint and attached advice have been executed in the refactored program version) must be equal to the original values. To implement our analysis, we built our system on the Java refactoring framework available in Eclipse, and used human interaction if we could not derive a result.

We conducted three case studies to evaluate our tool, one based on the source code of DynAMiT itself, one by analyzing the Jakarta Commons Codecs project, and finally one by analyzing the ANTLR parser generator framework. For each system, we analyzed if it is possible to semi-automatically refactor aspects from the automatically derived results presented by DynAMiT.

We soon discovered that DynAMiT–as a dynamic analysis tool–has two important disadvantage for automatic refactoring. Repeated call sequences are analyzed without any structural information about the underlying calls. This means that DynAMiT also reports repeated patterns if a call is governed by an `if`-statement or part of a loop, i.e. the found patterns strongly depend on the

[1]DynAMiT analyzes call relations without human interaction to derive candidates for crosscutting concerns. We use the term *automatic aspect mining* for comparable non-interactive techniques.

Case Study	DynAMiT						CommonCodecs						ANTLR					
Algorithm	Crosscutting			Basic			Crosscutting			Basic			Crosscutting			Basic		
Refactorizability	√	?	×	√	?	×	√	?	×	√	?	×	√	?	×	√	?	×
before call	0	0	2	0	0	3	0	0	0	0	0	2	2	0	6	1	0	43
after call	0	0	1	0	0	4	0	0	0	0	0	3	0	0	11	0	1	49
before execution	0	0	2	1	0	4	2	0	0	4	0	2	4	1	5	22	1	44
after execution	0	1	2	1	2	3	0	2	0	1	4	0	3	7	16	12	9	118
Coverage (statement)	56,9 %						96,8 %						25,7 %					

√ Semantics Preserving Refactoring feasible, × Refactoring failed due to Dependences, ? Semantical Change depends on Method Call

Table 1: Some Numbers: Candidates discovered by DynAMiT and their Refactorizability using our Tool

test suite used to generate the traces. If a insufficient test suite is used[2], extraction in an aspect is only possible and meaningful in the traces cases, and will produce different system semantics otherwise. This could be prevented if complicated pointcut expressions using the `if`-designator are generated to create functionally equivalent aspects. Consequently analysis of *control dependences* is important to check if an aspect candidate can be automatically refactored. However, we refrained from extracting such aspect candidates as such complex conditions more likely are an indicator for false positives (no quantified statement).

Second, method calls are not the only statements. We often experienced situations, where several assignments preceded the first call in a method. In these cases, the first call can only be extracted in a `before execution` advice if it is guaranteed that the joinpoint context is not modified by the above assignments, i.e. program semantics have to be equivalent if the call is moved to the method entry (code motion problem). That means *data-flow constraints* can considerably restrict refactorizability of crosscutting code.

Third, necessary joinpoint context sometimes is simply not available as no respective joinpoint exists in the target language (e.g. local variables or literals used in a call we want to extract in an aspect are not available for AspectJ). This means that *language limitations* also hinder aspect-oriented refactoring.

Please note that we did not examine the results of DynAMiT for *semantical soundness*, but only examined if they allow automatic refactoring resulting in a semantically equivalent program. For our case studies, most results had to be discarded. Table 1 gives some details on the case studies we performed. Consider for example the first column group labeled DynAMiT. Here, we got 8 aspect candidates in total when using the more strict "Crosscutting" algorithm (some of them symmetric). From these, only 1 candidate could be semi-automatically refactored. Our system has no pointer-analysis to safely approximate the effects of method calls, and thus does not allow us to automatically decide about refactorizability in all cases. We thus ask the user in such cases, using the *refactoring view* known from the Eclipse Java refactorings. These cases are counted in the '?' column. For the other 7 cases our tool found direct control and data flow dependences or was not able to access the context, all of which prevented refactoring. Note that DynAMiT is based on dynamic analysis, and thus the test coverage of the suite the analysis is based on is a very important issue in this context. The last line thus reports the coverage of the test suites we used for our case studies.

The Commons Codes system produced similar results. For ANTLR we got 55 advice candidates, and could only refactor 9 of these results. However, in 8 other cases a refactoring might have been possible, although our limited analysis could not decide this automatically. To summarize the above observation, control and data flow properties as well as language limitations are very important to decide if automatic refactoring of a crosscutting concern is possible. Recent work [2] of the author of DynAMiT also recognized these problems and added additional static analysis support.

To connect this observation with aspect-mining tools used for refactoring, analyzing and reporting data and control dependences can be used to (i) reduce the false positive rate and (ii) give additional information useful for programmers when they actually refactor code. Hence, analyzing refactorizability of candidates could be an additional criterion for the quality of an aspect-mining tool and might serve to give additional feedback to the user.

A second observation is that automatic syntax-based aspect mining tends to produce many false positives. Such tools *identify crosscutting code*–but crosscutting code not necessarily is due to a *crosscutting concern*. Crosscutting code is an indication for a crosscutting concern, but not a sufficient criterion; the decisive criterion is the *actual semantics* of the crosscutting code. Additionally the user still has to figure out all those identified patterns that actually belong to the same concern manually and thus should be encapsulated in the same aspect. So there is also a *mismatch in granularity* for purely syntax-based automatic techniques.

DynAMiT reported several candidates where we could not identify an semantical concern inducing the crosscutting code. From our perspective it is very hard if not impossible to distinguish between "accidentally" crosscutting code and crosscutting code due to an actual crosscutting concern without additional semantical information. This seems to be a general restriction of automatic syntax-based mining approaches.

3. MANUAL AM USING FEAT

Compared to the above study with automatic aspect-mining based on DynAMiT, we used a semantics-driven manual approach for HSQLDB. To find aspects here we based our analysis on the list of 'standard aspects' introduced in Laddad's book "AspectJ in Action" [5] and then used FEAT to discover relevant code locations. When becoming familiar with the source code we also found some *application specific aspects*, for example trigger firing or checking constraints before certain operations are performed.

To support manual system analysis, FEAT proved to be very effective. FEAT is a user guided cross referencing tool and allows to quickly discover code locations referencing some method or field. What we basically did–slightly simplified–was to discover potentially interesting classes–like e.g. `Tracing` or `Cache`–and then to use FEAT to discover where these classes are referenced. These references then have to be eliminated and replaced with an aspect to conduct the aspect-oriented refactoring.

We have finished the aspect mining phase and begun to actually implement the aspect-oriented refactoring. Our observations

[2]Note that for semantics-preservation, even a statement coverage of 100 % is *not* sufficient.

reported here are thus based on the aspects we identified, but we can not yet report if the aspect-oriented refactoring will actually be successful[3] in all cases. Refactoring in this case is manual, not automatic. Thus we are not as restricted in the ways we can refactor a system as in the above tool project.

For our analysis we manually discovered the starting point for a search, but this manual analysis was guided by our aspect catalog. We then used FEAT to find the locations where a crosscutting concern is tangled with other modules. Thus instead of using syntactical or low level properties of the system, we used a semantical approach. We started with a certain concern we expected to find in the system in mind, and tried to retrieve the code locations for its implementation. Compared to the above automatic study per definition no semantically questionable aspect candidates can occur. While this reduces the false positive rate, a considerable amount of *manual code inspection and analysis* (although supported by FEAT) was necessary to fulfill this task.

However, even for these manually identified crosscutting concerns, refactoring in general is not straightforward. Some of the problems we encountered are similar to the problems we discovered in the DynAMiT case study. FEAT also discovers calls to be extracted within a loop, or governed by an `if`-statement. Although this crosscutting code results from an *actual crosscutting concern*, refactoring these calls is nevertheless problematic. One strategy we use in this case is to pre-process the code (i.e. extract some code in methods if this is adequate) to allow a subsequent aspect-oriented refactoring. From a software engineering point of view, this cannot be a general solution as it easily jeopardizes system structure.

Our semantic catalog-guided approach was successful in discovering many standard aspects in the HSQLDB code base, including Tracing, Caching and Authentication/Authorization. To summarize, we think that augmenting aspect-mining tools with semantical information might be a fruitful approach for aspect mining. For example one might identify a set of classes related to a semantical concern–like e.g. a `Logger` class–and then demand that all reported candidates have to be related to one of these classes.

We will also illustrate these observations—both for aspect mining and refactoring—with an example. For HSQLDB, we identified the *tracing concern* as a crosscutting concern and its implementing crosscutting code. Tracing has been considered a standard crosscutting concern since the invention of AOP, so the aspect mining for this specific concern was relatively easy and straightforward following the strategy described above. Refactoring this concern and extracting its code into an aspect however was far from trivial. The problem is that *custom tracing* in an existing system cannot be formulated with a quantified statement like "On each method entry, log the method name and the parameter values.". The *implementation is* rather considerably more *customized for each method* to capture the values of interest within this method—including local variables and their changes e.g. within loops. Such customized tracing policies are very hard to capture in an aspect. We did this as an example for some classes, and to succeed we had to: create a *common trace format* (i.e. the system now produces a different output!), refactor *loop bodies to helper methods* (arranged pattern problem!), or *"promote" local variables to fields* (locality?). To make a long story short: the resulting implementation is—from a software engineering point-of-view—at least *questionable*.

However there are also positive examples. We identified *pooling*, also a standard crosscutting concern according to Laddad, as an aspect that can be refactored easily without the problems mentioned above. The source of HSQLDB contains a class `ValuePool` which contains relevant pooling logic. When an `Integer`, a `Long`, `String`, `Double`, `Date`, or `BigDecimal`-object is needed, the corresponding access method in the pool is explicitly invoked. Calls to these accessors occured at approximately 250 locations in the source code. As a result of these scattered calls we observed a high coupling between the classes containing these calls and class `ValuePool`.

For refactoring, these explicit calls to the value pool were replaced by the corresponding constructor calls (e.g. `new Integer()`). We then advised the constructors with `around`-advice which invokes the appropriate pool methods without calling `proceed`. This approach has several advantages compared to the purely object-oriented variant: As the pooling aspect is implemented as a separate aspect, the coupling due to the explicit pool invocations has disappeared (only the pooling aspect knows about the relation between class `ValuePool` and the remaining system). Second, the aspect now can be removed from the core program without any additional base changes. Finally, the aspect captures additional 190 code locations that failed to invoke the value pool before, as we used wildcards for the respective constructors to specify the pointcuts. To summarize, the aspect-oriented implementation in this case is clearly superior compared to the original version.

Although not all *refactorings* were successful, our semantic catalog-guided *mining* approach was nevertheless very successful in *discovering* many standard aspects in the HSQLDB code base, including Tracing, Caching, Pooling and Authentication/Authorization. To summarize, we think that augmenting aspect-mining tools with semantical information might be a fruitful approach for aspect mining.

4. LESSONS LEARNED

Most automatic aspect-mining approaches we are aware of are either based on finding repeated patterns in call sequences/traces/etc. or on finding duplicated code.

From our experience, actually refactoring advice candidates found by such tools has to deal with several important problems.

Control Dependences: Control dependences can easily lead to false positives, for example if a method call is always triggered in an available test suite used for analysis, but not necessarily triggered every time. While in some cases candidate code governed by an `if`-statement can be refactored to advice using an equivalent `if` pointcut designator, this is not true in general. Loops are an even more important problem.

Data Flow Restrictions: Advice cannot be attached to arbitrary code positions. If code should be moved to an aspect, it is possible that this code has to be moved e.g. to the beginning or the end of a method. This is of course not possible in general.

Arranged Pattern Problem: The code to be refactored in general uses some values from its context. These values thus always have to be accessible for the aspect language in order to allow a refactoring. Especially for AspectJ this is often problematic.

It is tempting to argue that any refactoring is possible, if we only use enough purely object-oriented refactorings to remove problematic control and data-flow dependences and make necessary joinpoint context available to our aspect language. However, this will result in another problem called the *arranged pattern problem* in [4]. Code is transformed only to allow advice application, but *not* to create well-defined, easy to understand, reusable, and evolvable methods. As a consequence, software quality degrades. This might be a language problem rather than an aspect mining issue, however suggesting such refactorings is problematic nonetheless.

Semantics vs. Syntactical Properties: The most important question: *Did we really find a crosscutting concern?* Even if

[3]I.e. if it is possible to refactor an identified concern or if the concern is too tightly coupled thus preventing refactoring.

syntactically a tool can derive an aspect candidate, is this candidate also semantically a valid concern? The use of *utility classes* is a good example. In general functionality of such classes is called from several parts of the system, however the modularity of the system is fine and nobody would argue that references to `java.lang.Math` show a crosscutting concern.

When looking at aspect mining results it is tempting to dismiss non-refactorizable candidates as false positives, although this is not true in general. However, if a crosscutting concern has an implementation too tightly coupled with the system, refactoring may not be a valid option anyway.

So is a purely semantic approach as we used for HSQLDB the method of choice? This method clearly has the advantage that we do not have to deal with many false positives. However, refactoring the code to advice faces the same problems as the automatic syntax-based aspect mining tools before. This justifies that simply removing non-refactorizable candidates from a result set is not a valid option, i.e. *'refactorizability' is no criterion to rule out candidates*—but it might help to order them by relevance for refactoring.

Semantic-based aspect mining also has another important disadvantage: We started our analysis based on the standard aspects catalog provided by Laddad. By only following this technique we will per definition only find aspects we know about–but *never new ones*. This is clearly a strength of syntax-based mining tools.

As a challenge to the aspect-mining community it might be interesting to create aspect mining tools which help programmers to identify standard crosscutting concerns in a given system. Of course such a tool could not be automatic, but compared to FEAT more automation might considerably help programmers trying to refactor standard crosscutting concerns. The main improvement we suggest is to also use semantical information to guide automatic tools when retrieving aspect candidates. Not all repeated method call sequences are advice candidates, but maybe those referencing a certain class are. Not each piece of duplicated code is a unrefactored advice, but maybe code referencing certain fields. This approach would combine automatic support from automatic aspect mining with the semantic guidance useful to avoid false positives.

While such a tool is interesting for a practitioner in the field trying to refactor an existing application based on a catalog of known aspects, it might also be interesting to develop tools designed to identify new aspects. Theses tools however are designed for researchers, who set out to better understand the nature of aspects in general, and also to extend the aspect catalog.

For evaluation of aspect-mining, both suggested tool categories have a considerably different profile and need different evaluation strategies. Tools targeted to discover standard aspects need appropriate systems where a refactored aspect-oriented and an original version exist. Based on these two versions, quality of the results is accessible. Evaluating tools designed to discover new aspects is considerably harder. The above strategy is not useful in this case.

5. CONCLUSION

In this paper we discussed the results of a fully automatic aspect mining tool in contrast to a manual aspect mining approach.

From our experience many aspect candidates proposed by the automatic aspect mining tool are not useful for an automatic refactoring, as language restriction, i.e. un-accessible context, control dependences, i.e. calls to-be-extracted which are embedded in loops/governed by `if`-statements, or data-dependences, i.e. modifications of parameter values prior to calls to-be-extracted, prevent refactoring. Reviewing these problems for particular cases often also raises doubt if the corresponding code is actually part of the implementation of a crosscutting concern.

Our second study based on manual aspect mining was successful to discover standard aspects, but failed to reveal any new/application specific aspects. This is a general weakness of this approach. While here per definition no false positives occur (either a valid concern can be found or not), refactoring the found crosscutting code might not be recommendable due to a high coupling with the base system.

To improve result quality for aspect mining tools, we suggest to build two kinds of tools: (i) aspect mining tools guided by a catalog of well-known crosscutting concerns to assist software engineers in actually refactoring existing systems and (ii) less restricted automatic mining tools designed to help researchers find completely new aspects. For the first category of tools refactorizability might be a good criterion to prioritize mining results.

Using projects like AJHotDraw and HSQLDB as case studies (once our project is finished) seems to be a good way to evaluate category (i) aspect-mining tools. We encourage researchers to use their tools to also refactor other projects as case studies and make the resulting aspect-oriented systems publicly available.

Acknowledgments

Thanks to the anonymous reviewers and Daniel Wasserrab for their valuable and interesting comments on this paper.

6. REFERENCES

[1] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated Refactoring of Object Oriented Code into aspects. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 27–36, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Silvia Breu. Extending Dynamic Aspect Mining with Static Information. In *5th International Workshop on Source Code Analysis and Manipulation (SCAM 2005), Budapest, Hungary*, October 2005.

[3] Silvia Breu and Jens Krinke. Aspect Mining Using Event Traces. In *19th International Conference on Automated Software Engineering (ASE 2004)*, pages 310–315, September 2004.

[4] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM Press.

[5] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[6] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, New York, NY, USA, 2002. ACM Press.

[7] Stefan Schöffmann. Semi-automatisches Aspect Refactoring–Tool-Entwicklung und Fallstudie auf Basis bestehender Aspect Mining Tools. Master's thesis, Universität Passau, Innstraße 32, 94032 Passau, Germany, Dezember 2004.

[8] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[9] Arie van Deursen, Marius Martin, and Leon Moonen. AJHotDraw: A showcase for refactoring to aspects. In *In Proceedings AOSD Workshop on Linking Aspect Technology and Evolution*, 2005.