2008 15th Working Conference on Reverse Engineering

Pitfalls in Aspect Mining

Kim Mens
Université catholique de Louvain, Belgium
kim.mens@uclouvain.be

Andy Kellens
Vrije Universiteit Brussel, Belgium
akellens@vub.ac.be

Jens Krinke
FernUniversität Hagen, Germany
krinke@acm.org

Abstract

The research domain of aspect mining studies the problem of (semi-)automatically identifying potential aspects and crosscutting concerns in a software system, to improve the system's comprehensibility or enable its migration to an aspect-oriented solution. Unfortunately, most proposed aspect mining techniques have not lived up to their expectations yet. In this paper we provide a list of problems that most aspect mining techniques suffer from and identify some of the root causes underlying these problems. Based upon this analysis, we conclude that many of the problems seem to be caused directly or indirectly by the use of inappropriate techniques, a lack of rigour and semantics on what is being mined for and how, and in how the results of the mining process are presented to the user.

1 Introduction

Several years after the emergence of the aspect-oriented software development paradigm, some researchers recognised the need for and relevance of reverse engineering techniques in the context of aspect-oriented software. The new research domain of aspect mining was born. Just like the adoption of any new programming paradigm evidently leads to the question how to migrate existing systems to that new paradigm, the same happened to aspect-oriented programming. The research area of *aspect mining* addresses part of this problem, namely the identification of potential aspect candidates in the source code of existing software systems [8, 16]. A second part of the problem, addressed by the *aspect refactoring* community, is how to transform the identified aspect candidates into actual aspects in the software [4, 23]. Unfortunately, most of this research is still relatively immature and most of the proposed techniques have not lived up to their expectations yet.

In this paper we focus on the domain of aspect mining in particular, and try to identify the root causes of why the proposed techniques fail to deliver. To this end, we base ourselves on our own experiences in aspect mining research, and on an extensive survey of existing aspect mining techniques we conducted. Our first goal is to provide an extensive list of problems encountered with currently existing aspect mining techniques, based upon our own experience and that of others. Our second goal is to try and identify the main underlying causes of these problems. Thirdly, based on our analysis of the problems and their main causes we discuss the future of aspect mining: is there a future, what can be solved, what can be improved?

This paper presents a moment of reflection on the state of research in aspect mining. Most identified issues are not new or unknown, nor are they unique. They occur scattered throughout recent aspect mining literature, and the reverse engineering community has faced similar problems. The merit of this paper is in collecting these issues and to provide a broader basis for discussion on the topic.

2 Aspect Mining

Crosscutting concerns have always caused problems in software development and its maintenance because of the scattering and tangling of concerns' implementations [27]. Aspect-oriented software development has been introduced to cope with these problems by encapsulating crosscutting concerns into a new abstraction called *aspects*. However, there are several research questions that still require investigation. One of those questions, when migrating a non aspect-oriented system to an aspect-oriented one, is how to identify those crosscutting concerns that can be turned into aspects. We call those crosscutting concerns *aspect candidates* and the activity of identifying them *aspect mining*.

Our survey paper [16] distinguished three different categories of techniques that could help in locating the aspect candidates in a software system: early aspect discovery

1095-1350/08 $25.00 © 2008 IEEE
DOI 10.1109/WCRE.2008.10

techniques, dedicated code browsers and automated aspect mining techniques. *Early aspect* mining techniques identify and manage crosscutting properties from the early software development stages [2] of requirements and domain analysis [1, 24, 28] and architecture design [3]. *Advanced special-purpose code browsers* (like Concern Graphs [25], Intensional Views [22], Aspect Browser [12], (Extended) Aspect Mining Tool [14, 31], SoQueT [20] and Prism [32]), on the other hand, aid a developer in manually navigating the source code of a system to explore potential aspect candidates. These techniques typically start from a so-called *aspect seed*, a location in the code from which the users start their navigation to explore other places in the code which might be part of the same concern. This way, the user iteratively constructs a model of the different places in the code that correspond to an aspect candidate.

Complementary to the early aspect mining techniques and dedicated browsers, *automated aspect mining techniques* aim at automating the aspect identification process and proposing one or more aspect candidates with as little user intervention as possible. These automated aspect mining approaches are the ones we will focus on in this paper. They all have in common that they reason about the system's source code or execution traces and that they search for symptoms of crosscutting concerns. Typically, they use either techniques from data mining and data analysis like formal concept analysis and cluster analysis, or more classic code analysis techniques like program slicing, software metrics and heuristics, clone detection and pattern matching techniques, dynamic analysis, and so on.

In [16] we conducted an extensive survey of aspect mining techniques which semi-automatically assist a developer in the activity of mining potential aspect candidates from the source code of an existing system. We observed a lot of variation in those techniques, depending on what underlying techniques they rely upon (e.g., clone detection, clustering, concept analysis), the kind of analysis they perform (e.g., static or dynamic, structural or behavioural), the granularity of the results being reported (e.g., entire methods, code fragments), what underlying assumptions the techniques make about the program being analysed (e.g., size, use of coding conventions, presence of code duplication, existence of use cases), what symptoms of crosscutting concerns they look for (e.g., scattering, tangling, code duplication), the kind of user involvement required (e.g., preprocessing of input data, post-processing of results), and on how well the techniques have been validated.

For a more detailed overview and comparison of those techniques we refer to that survey paper. In this paper, having noticed that each technique has its own limitations and weaknesses, we build on that paper and list the main problems we observed in current-day aspect mining techniques, before trying to understand the main causes behind those problems and suggesting ways of solving or avoiding them.

3 Problems with aspect mining

In this section we present a list of typical problems that current aspect mining techniques suffer from. To structure the discussion, we present these problems in a pattern-like format consisting of the name of the problem, a short description, a concrete illustration of the problem and (optionally) a set of other, related problems.

3.1 Poor precision

Description Many current-day aspect mining techniques exhibit poor precision, meaning that the percentage of *relevant* aspect candidates in the set of all candidates reported by a given technique is relatively low. While this low precision is not a problem *in se*, it does imply that aspect mining techniques tend to return a lot of false positives, which can be detrimental to their scalability and ease-of-use. Especially for techniques that return a large number of results, this lack of precision can be problematic, since it may require an important amount of user involvement to separate the false positives from the relevant aspect candidates.

Note that precision can be observed at several levels of granularity:

1. At the level of *individual* aspects or concerns: do we find some things that are not aspects or concerns?
2. At the level of *code*: for a given aspect candidate or seed we detected, are the code fragments we find as belonging to that concern really a part of that aspect?
3. At the level of *crosscutting sorts*[1]: if we look for all aspects or concerns *of a given kind*, how many false positives do we find that do not belong to that kind?

In most papers describing aspect mining techniques, if they report on the precision of the technique at all, typically only one kind of precision is calculated (depending on the granularity of the results produced by the technique).

Example Bruntink et al. [6, 7] evaluated the suitability of clone detection techniques for automatically identifying crosscutting concern code. They considered 16,406 lines of code belonging to a large industrial software system and five known crosscutting concerns that appeared in that code: memory handling, null pointer checking, range checking, exception handling and tracing. Before applying their clone

[1] A crosscutting sort is a class of crosscutting concerns that share similar properties such as intent, behaviour, and so on. For example, concerns of the sort 'contract enforcement' and 'consistent behaviour' generally describe common functionality implemented by many crosscutting methods, such as a specific pre-condition check on certain methods in a class hierarchy [19].

Technique: / Concern:	AST	Token	PDG
Memory handling	65%	63%	81%
Null pointer checking	99%	97%	80%
Range checking	71%	59%	42%
Exception handling	38%	36%	35%
Tracing	62%	57%	68%

Table 1. Average precision of each technique for each of the five concerns

detection techniques to mine for the code fragments (lines of code) belonging to each of those concerns, they asked the developer of this code to manually mark, for each line of code, to what concern(s) it belonged. Next, they applied three different clone detection techniques to the code: an AST-based, a token-based and a PDG-based one. In order to evaluate how well each of the three techniques succeeded in finding the code that implemented the five crosscutting concerns, the results of each of the clone detection techniques were compared to the manually marked occurrences of the different crosscutting concerns, and precision and recall were calculated against those. Table 1 shows the average precision of the three clone detection techniques for each of the five concerns considered.

As can be seen from the table, the results of this experiment were rather disparate. For the null pointer checking concern, all clone detectors identified the concern code at near-perfect precision. For most of the other concerns, none of the clone detectors achieved satisfying precision.

In a way, the experiment conducted by Bruntink et al. [7] was a kind of ideal situation because they actually compared the results of their technique with the crosscutting concerns marked by a developer in the code. In absence of such documentation on the crosscutting concern code, most other aspect mining experiments either fail to report on the precision of their technique, or they compare the results with their own judgement (which may lead to subjectivity). Yet, even such experiments most often report a disappointingly low precision.

Related problems As will be explained in the forthcoming subsections, poor precision has a negative impact on scalability (3.4) and may be due partly to the problem of subjectivity (3.3). There is also a subtle trade-off between recall (3.2) and precision: often better precision can be reached at the cost of lower recall and vice versa.

3.2 Poor recall

Description Recall is the proportion of *relevant* aspect candidates that were discovered out of all aspect candidates present in the source code. In other words, recall gives an idea of how many false negatives remain in the code and thus how well (or not) the technique covers the entire code analysed. As for precision, recall can be observed at several levels of granularity. At the level of individual aspects or concerns: do we find all aspects and concerns that are present in the code? At the level of code: do we find the full extent of the aspect or concern or does the technique fail to discover some code fragments pertaining to the aspect? At the level of crosscutting sorts: if we look for all aspects or concerns of a given kind, do we find all concerns of that kind which exist in the code?

A problem with calculating recall is that typically, in a program under analysis, it is not known what the relevant aspects and code fragments are, except in an ideal case like the validation experiment of Bruntink et al. (see above) where the concerns are known in advance and where a programmer took the time to mark each line of code with the concern(s) it belongs to. A second problem is that most techniques will look for certain symptoms of aspects only and thus are bound to miss occurrences of aspects that exhibit different symptoms.

Examples In their aspect mining experiment, Zhang and Jacobsen [33] report on a recall of only slightly above 50% (with a precision of slightly above 70%). A similarly low (and sometimes lower) recall is reported for most other aspect mining techniques we are aware of, if they report on recall at all. In addition, as was the case for precision as well, most authors typically calculate only one kind of recall and do not mention explicitly at what level of granularity the recall was considered.

As a more detailed example of recall at the level of crosscutting concerns, consider the following example taken from Ceccato et al. [8]. By comparing the results of three different aspect mining techniques, the authors observed that some techniques where better at finding instances of some crosscutting sorts than others. For example, consider the 'contract enforcement' or 'consistent behaviour' sorts mentioned in footnote 1. An example from the JHotDraw application is the Command hierarchy for which the execute methods contain code to ensure the pre-condition that an 'active view' reference exists (is not null). Since such consistency checks are typically implemented by having the methods call the same auxiliary methods, the technique of fan-in analysis [21] proved to be particularly suited at finding instances of these kinds of concern. Indeed, since fan-in analysis looks for methods with a high fan-in, it easily finds those auxiliary methods that are called by many other methods. A technique like identifier analysis [30], however, which mainly looks for crosscutting methods with a similar naming scheme, only found some instances of that concern sort but missed those instances where the methods that enforce a given contract or ensure consistent behaviour did not share a common naming scheme.

On the other hand, the technique of fan-in analysis

scored worse at the level of code fragments. Indeed, by its nature the technique only discovers methods with a high-fan in. These aspect seeds, if relevant, must then be completed by the user to include many other methods calling or called by those methods, that are part of the aspect as well.

Related problems As mentioned before, recall often has an inverse correlation to precision (3.1) as a higher recall tends to cause the precision to decrease, and vice versa. Subjectivity (3.3) can also have an impact on recall as it may cause relevant concerns to be missed. Finally, there is a link with scalability too (3.4) since the larger the system being analysed is, the harder it will be to calculate the recall (especially since there exists no automated means of calculating recall).

3.3 Subjectivity

Description For many existing aspect mining techniques, the produced results exhibit some ambiguity. Depending on the person and the definition of aspect he or she uses, sometimes one person would say that something is an aspect candidate, whereas another person would say that it is not.

Example In their comparison of three different aspect mining techniques (i.e., *identifier analysis*, *use case analysis* and *fan-in analysis*), Ceccato et al. [8] observed a certain amount of subjectivity in the interpretation of the results proposed by the different techniques. First of all, every tool uses its own specific filters to discard certain results and keep others, and some of these filters can be partly configured by the user (e.g., by setting a certain threshold value or by ignoring certain input). Secondly, in the list of filtered results, the user typically has to make a selection of which results do and which do not represent valid aspect seeds or aspect candidates. This user involvement may be a cause of subjectivity, even for those cases where a similar concern is proposed by different tools. For example, when applied to the JHotDraw application, all three techniques flagged the 'Moving figures' concern as a potential aspect. The researchers conducting an *identifier analysis* as well as those performing a *use case analysis* considered this concern as a potential aspect and included it in the list of relevant results. The team which used *fan-in analysis*, however, discarded the concern, arguing that the original design seems to consider this functionality as part of the application's core logic of handling figures. This shows that, depending on the definition of aspect that is explicitly or implicitly used, the analysis may lead to different results, even when the brute results produced by the tools are the same.

Related problems Subjectivity may have a negative impact on the quality of the results produced by a technique, and on precision (3.1) and recall (3.2) in particular. Any kind of empirical validation (3.5) suffers from subjectivity.

3.4 Scalability

Description An important property of any given aspect mining technique is its *scalability*. One factor that has an impact on scalability is the *time-efficiency* of the tool, i.e., the amount of time required for the tool to compute its results (how long it takes for the tool to run). Most tools do not seem to be problematic in this respect. Another factor contributing to *scalability*, however, is the amount of *user involvement* required for a given technique. Often, the time required for an aspect mining tool to calculate its results, is negligible with respect to the amount of time required for a tool user to pre-process the tool's input and/or post-process and analyse its output.

Example Although the problem of user involvement holds for several known aspect mining techniques, identifier analysis [30] is a technique that suffers in particular from this scalability issue.

The identifier analysis technique performs a formal concept analysis to group methods with similar names. To increase the precision and limit the number of results returned by the analysis, a number of different filters are applied. One filter restricts the search to those concepts that exhibit a certain amount of crosscutting (by checking that the methods grouped in such a concept belong to different class hierarchies). Other filters reject concepts that are too small (too few keywords in common or too few methods in the concept) and ignore methods of which the names contain keywords belonging to an experimentally determined blacklist.

As reported by Ceccato et al. [8], applying the identifier analysis technique on JHotDraw yielded 230 concepts and took about 31 seconds. This was when a threshold of 4 for the minimum number of methods in a reported concept was used (i.e., when applying a filter that refuses all concepts containing less than 4 methods sharing similar keywords in their identifier). When conducting the experiment again with a stricter threshold of 10 for that filter, the number of concepts produced was significantly reduced: only 100 concepts remained after filtering, for a similar execution time. Applying this stricter filter thus removed a lot of the noise produced by the less strict filter, yet without loosing too much interesting results. Nevertheless, given that every concept grouped about 6 methods on average, the amount of data to browse through by a user to validate the results was still significant. (Note that with a threshold of 4, not only were there many more concepts, the average number of methods per concept was higher as well: 17 instead of 6). Therefore, in their comparison, Ceccato et al. [8] concluded that the identifier analysis technique is probably more useful as a technique to complement the sometimes partial results proposed by other techniques, than as a standalone technique.

Some tools also require certain user involvement to provide the tool with the appropriate input. This is for example the case for the *use case analysis* technique [29] which takes as input a set of use cases for the analysed software system. Providing the tool with a relevant and appropriate set of use cases is the tool user's responsibility. For large applications, this may require a significant amount of work.

3.5 Empirical validation

Description It is impossible for the discipline of aspect mining to make further progress without *empirical validation* of the results. However, validating the quality of an aspect mining technique is an intrinsically difficult problem. A good empirical validation of aspect mining techniques requires the ability to measure the precision and recall of the results, at different levels of granularity. The subjectivity of the interpretation of the results, however, obstructs such an empirical validation: analysing the results of a technique or comparing it with the results of a different technique becomes subject to the researchers performing the experiment, thus limiting the reproducibility of the results.

Furthermore, to demonstrate the scalability of the approach, user studies should be conducted. End users of aspect mining techniques (e.g., the programmers of a system on which the aspect mining is being applied) should be involved in order to evaluate the actual usefulness and usability of each proposed technique.

Example Most of the approaches we studied do not provide an empirical validation of their results but rather provide a more incidental validation of their work. They demonstrate how particular interesting crosscutting concerns can be identified using a specific technique. While this can give an indication of the adeptness of the technique for identifying crosscutting concerns, this does not provide any quantitative information nor a sufficient basis for objective comparison of techniques.

One example of a successful empirical validation of an aspect mining technique is the work of Bruntink et. al [7] which we already discussed in Subsection 3.1. In their work, they were able to compare the results of the mining process with a version of the system that was manually annotated by the original developers, thus making an empirical validation possible. In general however, a common benchmark for aspect mining techniques, containing a complete and sound set of aspects over which there exists a consensus is missing.

3.6 Other problems

In this subsection, we briefly mention some other problems related to aspect mining, but do not discuss them in detail due to space limitations.

Comparability It has shown to be difficult to compare the results of different techniques. There are different causes to this problem: difference in granularity (level of detail) of the results, difference in kinds of results (sometimes just method names, sometimes code fragments, sometimes just methods being called, ...), subjectivity in interpretation of the results, etc.

Composability Because of the observed limitations of some aspect mining techniques, there is a desire to *combine* the techniques of different researchers [8]. However, for reasons similar to those mentioned just above, this is not that easy in practice.

Simple crosscutting concerns are not so simple In [5], the implementation of the tracing concern in a large industrial case was studied. Although tracing is traditionally considered as a "simple" crosscutting concern, it turned out that the idiom programmers used to implement this concern exhibited remarkable variability. This variability significantly hinders the task of automatically mining instances of that concern from the code. Although part of this variability is accidental and due to typing errors or improper use of idioms, a significant part of the variability turned out to be essential. Therefore, aspect mining techniques should explicitly take this variability into account to mine for relevant aspect candidates. If even with a "trivial" concern like tracing we already have such variability, things will probably get worse when more complex concerns are being investigated.

4 Analysis of the problems

By carefully analysing the above problems, we managed to bring them back to three main root causes: inappropriateness of the techniques used to mine for aspects, lack of a precise definition of what constitutes an aspect, and inadequate representation of the aspect mining results.

4.1 Inappropriate techniques

A first important cause of many of the observed problems is that most current-day aspect mining approaches use too simple techniques, not the right techniques, or the available techniques in a too simple way. Below we discuss a variety of reasons we identified why current aspect mining techniques are not well-suited at their job.

4.1.1 Too general-purpose

Most current aspect mining approaches rely on a *general-purpose* mining technique in order to identify aspect seeds or candidates. Consequently, such approaches may be too general-purpose and result in poor performance.

Firstly, some of the approaches we studied make use of traditional data mining techniques such as cluster analysis, formal concept analysis or natural language processing in order to group source-code entities that—according to some similarity criterion—might belong to the same crosscutting concern. In order to tweak these techniques to the particularities of aspect mining in general and the system that is being analysed in particular, a considerable amount of expertise is required to fine-tune and customise the parameters of the technique and the post-processing of the results. For example, the choice of thresholds, applied filters, and so on strongly depend on what particular technique is being used and to what system it is being applied.

Secondly, most current aspect mining techniques are not dedicated to finding instances of one particular crosscutting sort, but rather mine the source code of a system for *any* crosscutting concern. Due to the multitude of possible crosscutting sorts this assumption of a "one size fits all" aspect mining technique appears to be too broad.

Instead, dedicated techniques that look for particular kinds of crosscutting concerns may do a better job (with higher precision and lower recall) of finding valid aspect candidates because they can be fine-tuned to the particularities of the concern sort they are interested in.

4.1.2 Too strong assumptions

All aspect mining techniques make certain assumptions about how, for example, crosscutting concerns are implemented in the source code, to identify groups of source-code entities that exhibit the assumed symptom of crosscuttingness. Examples of such symptoms used by existing aspect mining techniques are recurring call patterns, code duplication and high fan-in values. By relying too hard on these assumptions, current aspect mining techniques tend to suffer from two problems.

On the one hand, some techniques are too dependent on or too tightly coupled with how the source code of a system is structured. They only look for crosscutting concerns that are revealed by a very specific way of how the concern itself and its embedding base code are structured. Consequently, these techniques will only detect those aspect candidates that match this particular assumption. Unless a particular aspect mining technique takes many different assumptions into account, this will result in that a given technique only finds a limited subset of the possible aspect candidates.

On the other hand, the assumptions made by a given technique about the implementation of a particular kind of crosscutting concern most often do not take possible *variations* in the crosscutting concern's implementation into account. As we already mentioned in Subsection 3.6, in legacy systems even seemingly simple crosscutting concerns are not always uniformly implemented. This lack of homogeneity in the implementation of aspect candidates affects the recall of aspect mining techniques. As such, even crosscutting concerns that deviate only slightly from the assumed implementation can be missed by a technique.

4.1.3 Too optimistic approaches

Most (if not all) techniques we are aware of are optimistic approaches: they only search for symptoms in the source code supporting the hypothesis that a code fragment is part of an aspect or a crosscutting concern, but do not look for symptoms supporting the anti-thesis that the fragment is *not* part of an aspect or crosscutting concern. A code fragment may exhibit all the right symptoms that make it look like it is part of an aspect or crosscutting concern, but this hypothesis may be invalidated by certain counter-arguments.

As an example of such a situation, consider the "Moving Figures" concern discussed in Section 3.3. This concern was discovered by the three techniques mentioned there and also by Krinke's *execution-relation analysis* [17]. Similar to *fan-in analysis*, Krinke's *execution-relation analysis* discovers it because a method *Rectangle.translate* is called in eight instances of a *basicMoveBy* method (supports the hypothesis). However, there are nine other instances of that method which call *Rectangle.translate* for different purposes or not at all (support of the anti-thesis). None of the four techniques uses or even presents this information to the user.

4.1.4 Scattering versus tangling

Two main symptoms of the presence of aspects are scattering and tangling. While *scattering* is the phenomenon that crosscutting code fragments tend to get *spread throughout* the entire system, *tangling* is the phenomenon that some cleanly localised core functionality may get cluttered with these crosscutting code fragments.

Almost all aspect mining techniques focus exclusively on detecting symptoms of *scattering*. While scattering certainly is an indicator of crosscuttingness, it alone does not suffice for correctly identifying valid aspect candidates. This problem was exemplified nicely by the poor precision and recall of some of the clone detection techniques mentioned in Subsection 3.1. It appeared that this was related to the amount of tangling of the concerns. Clone detectors achieved higher precision and recall for concerns that exhibited relatively low tangling with other concerns or with the base code, than for concerns that exhibited high tangling. We conclude that the symptom of tangling is at least as important to consider when mining for aspects.

Incorporating this notion of tangling into mining techniques however is far from trivial. In order to approximate tangling in the implementation, information about the different concerns that are present in the system is needed. Since this information is often hard or impossible to obtain, and is partly what we are mining for in the first place,

most current-day aspect mining techniques do not take this symptom into account.

4.1.5 Lack of use of semantic information

While relying on symptoms of crosscutting concerns (like scattering and tangling) makes it possible to identify potential candidate aspects, as discussed above it also results in the introduction of poor precision and recall. Although crosscutting concerns can often be characterised by a particular symptom, we also saw that this does not necessarily mean that all source-code entities exhibiting this symptom are part a crosscutting concern. For example, one wide-spread symptom of crosscuttingness is code duplication. While techniques that focus on finding some form of code duplication can positively identify crosscutting concerns, this code duplication can also identify non-crosscutting concerns. In addition, aspects do not imply crosscutting (depending on the definition of aspect that is taken)[2], hence these aspects are not discovered by most techniques (since most techniques seem biased to looking for symptoms of crosscutting).

Finally, without semantic knowledge it is hard to decide how a crosscutting piece of code is coupled to the embedding code. More semantic information on this coupling can help in deciding whether the structurally crosscutting piece of code is crosscutting in the aspect-oriented sense, and thus represents a potential aspect candidate. This is not necessarily the case. For example, as reported by Krinke [17], instances of delegation are by some techniques (accidentally) identified as being a potential crosscutting concern.

4.2 Imprecise definition

The second main cause of many of the problems we encountered with current-day aspect mining techniques is the *lack of a sound and precise definition* of what constitutes an aspect or a crosscutting concern. Without a clear and unambiguous definition of what we are mining for it is hard to define and validate appropriate mining techniques. While current-day techniques can identify the manifestation of useful aspect candidates in the source code of a system, the lack of a more formal definition causes the interpretation of the results obtained by these techniques to be *subjective* to the actual user of the technique. Consequently, this also affects the ease with which it is possible to perform *empirical validation* of an aspect mining technique: validating the correctness of a mining technique's results is impossible if it is not clear what it is exactly the technique tries to find.

Apart from the lack of a good definition there is the problem that many approaches seem to *equate crosscuttingness with aspects*. In general, aspect mining techniques return sets of source-code entities that—according to a particular definition—are crosscutting and proposes these to the user as aspect candidates. However, if the goal of the mining process is to find crosscutting concerns that can be extracted into aspects, this vision on aspect mining seems to be too broad. When a particular piece of the code is identified as belonging to a crosscutting concern, this does not necessarily imply that this piece of code can or should be extracted into an aspect. For example, many of existing techniques' results include trivial crosscutting concerns for which it is not interesting to extract them from the source code. Similarly, as reported by [4, 23], the lack of a clear structural pattern in the source code of a crosscutting concern or the way it is implemented may require that classic object-oriented refactorings of the code are necessary in order to extract it, or that extraction into an aspect is not feasible.

The other problem with equating crosscuttingness with aspects, as already mentioned in point 4.1.5, is that aspects that do not have a crosscutting nature may be overlooked.

4.3 Inadequate representation of results

A final problem category is related to the representation used for the results of an aspect mining technique. First, the *level of granularity* at which the results of a technique are represented can impact the quality and usability of the technique in several ways. If a too coarse granularity is used, this may make it harder for users to discern whether a proposed concern is indeed a valid candidate aspect. Providing too much detail on the other hand can result in the user being overwhelmed by the amount of data he needs to process in order to filter out the irrelevant information. However, if the results of the mining process need to serve as input of a extraction step, sufficient detailed information concerning the joinpoints and their context should be returned.

In addition, not only the level of granularity, but also how the results are presented to the user (method names, callees and callers, shared code fragments, similar types used) may be a cause of ambiguity because they force the user's mind in a certain direction. Finally, there is no standard representation format for representing the results of aspect mining results, which makes it hard to compare or combine different aspect mining techniques.

4.4 Conclusion

We identified three main causes that seem to lie at the root of many of the analysed problems in Section 3. These causes are somewhat overlapping in the sense that even if two of these causes are taken away, some problems will still remain because of the third cause. For example, even if there would exist a precise definition of aspects and an

[2]For example, in AspectJ [18] it is easy to write an aspect that affects only a single class or even a single joinpoint. The language by no means imposes the aspect definitions to be crosscutting.

Cause:	*Inappropriate techniques*					*Imprecise definition*	*Inadequate representation of results*
	too general purpose	too strong assumptions	too optimistic approaches	no attention to tangling	lack of use of sem. info		
Problem:							
Poor precision	-	(+)	-	-	-	-	-
Poor recall	(+)	-	(+)	-	-	-	
Subjectivity					-	-	-
Scalability	(-)	(+)	(-)	(-)	(+) (-)	-	(-)
Empir. valid.						-	-
Comparability						-	-
Composability						-	-

Table 2. Aspect mining problems and their causes

adequate and standard representation of results, the use of an inappropriate technique would still lead to low quality results. A second observation is that, with the notable exception of the lack of consideration for the symptom of tangling, none of these causes seem specific to the problems encountered with aspect mining but are relevant to any kind of pattern mining technique. However, the problems are especially present for aspect mining due to the (still) relative immaturity of this research area.

Table 2 summarises the impact of the different causes identified above on the problems listed in Section 3. A '-' sign indicates a negative impact, most of which have been discussed above. The '-' sign is put between parentheses when the negative impact is less direct. A '(+)' sign indicates a positive impact. We put it between parentheses because our main focus is on the problems and their causes.

Having already discussed the most obvious negative impacts in the previous subsections, we distil our main findings here. First of all, we observe that all causes negatively affect precision, recall or both. Indeed, it is not surprising that limitations in either the technique, definition or representation of the results tend to lower the quality of the results. Secondly, the absence of a precise definition of what constitutes an aspect, the lack of use of semantic information as well as the inadequacy of how results are represented are potential causes for most of the observed problems.

Other negative impacts are less direct. For example, techniques with poor precision tend to be less scalable simply because they produce more false positives that need to be analysed by the user.

Finally, we can see from Table 2 that some causes may positively affect some observed problems. For example, in contrast to a dedicated mining technique that tries to find instances of certain kinds of aspects or crosscutting concerns only, a general-purpose technique may improve the recall. Too optimistic approaches may positively influence the recall as well, simply because they impose less restrictions. In contrast, a technique that makes very strong assumptions about the source code may have a higher precision, at the cost of a lower recall. This is the well-known trade-off between precision and recall. Scalability too may be positively affected by techniques that make very strong assumptions, simply because they limit the number of results produced. Since using more semantic information may require a more detailed source code analysis, the *lack of* using such information can positively influence scalability.

5 Discussion

In this final section we take a step back from the problems encountered with aspect mining techniques and our analysis of the causes of these problems, and distil some important lessons for future aspect mining techniques.

Concern-awareness A first important observation to be made is that the desired quality (e.g., in terms of precision and recall) of an aspect mining technique depends on what you want to achieve with it. If your goal is to get an initial understanding of the crosscutting nature of a software system, then poor precision and poor recall are not necessarily dramatic. If, on the other hand, your goal is to automatically migrate a non aspect-oriented legacy system into an aspect-oriented one through automated aspect mining and subsequent aspect refactoring, then a much more rigourous aspect mining approach is essential.

Before doing such a migration it is important to decide whether there is an actual need to do so. It is clear that applying an aspect mining technique can support system comprehension by identifying the crosscutting concerns in the source code and where they are located. But is it necessary to refactor the system in such a way that the crosscutting concerns are replaced by aspects? Are the crosscutting concerns present in the system really hampering software quality? Until now, there is insufficient support for the thesis that extraction of crosscutting concerns and replacement by aspects does improve the understandability, quality or maintainability of a system.

Consider the analogy with the problem of clone detection. For years, the presence of code clones has been considered as an obstacle to software maintainability, and research has looked at techniques to automatically discover

and refactor code clones and replace them by functions or macros. Today, the pendulum swings in the other direction again with recent publications claiming that the presence of clones alone is not necessarily bad [15] and that clone-aware development environments [9] that support the developer and maintainer could be a better approach. Similarly, aspect mining could focus on concern-aware environments [22] that allow developers to understand and document the concerns in their system without necessarily having to transform them into actual aspects.

Improved aspect mining techniques Probably the most important improvement would be the use of more intelligent mining techniques that go beyond the search for purely structural patterns in source code. Mining techniques could be made more appropriate by taking more semantic information into account, by looking at symptoms of tangling in addition to scattering, by taking negative information (support for the anti-thesis) into account, and so on.

Some authors have suggested to *combine* aspect mining techniques to improve the overall result. However, a mere combination of techniques may not suffice as it does not solve the underlying problems. Also, a combination is currently not easy because of the composability problem.

Paying more attention to how the results of a technique are presented may be beneficial for avoiding subjectivity in interpretation (and thus improving the precision), and for being able to validate the technique or to compare and combine it with other techniques.

Dedicated aspect mining techniques It is doubtful whether a single aspect mining technique can be conceived that identifies the complete range of aspects with high precision and recall. Future aspect mining techniques may be better of targeting specific classes of aspects, ranging from development aspects to complex production aspects.

To do so, they should be very precise on what kinds of aspects or crosscutting concerns they can and cannot identify. This will decrease the problems of subjectivity, scalability, validation, comparability and composability, and at the same time make it easier to fine-tune the technique so that it achieves higher precision and recall.

To allow for a methodological way of comparing and combining the large number of aspect mining techniques that have been proposed in literature, Marin et. al. [19] propose a common framework, based on crosscutting concern sorts, that allows for a consistent evaluation, comparison and combination of aspect mining techniques. Many of the problems listed in Section 3 were explicitly part of their motivation for developing such a framework.

Similarity with design pattern mining It is worthwhile to note similarities with design recovery tools [13] and *design pattern mining* in particular, where systems are mined for occurrences of (specified) design patterns. Most approaches in design pattern mining analyse a structural representation of a system for instances of specific structural patterns. Interestingly, authors report problems similar to the ones we have identified. Usually, the approaches have low precision and recall. A comparison of three design pattern mining tools [11] also revealed that, because design patterns are not defined formally, different tools found different design pattern instances mostly because of their different pattern definitions. Also, the use of different mining algorithms resulted in differences in the output. A final observation is that the need for more intelligent techniques has been recognised by the design pattern mining community too. An example of such an improved technique is the combination of design pattern mining with machine learning [10] to filter out false hits. Such an approach is promising for aspect mining too, because it enables to increase the precision by making the approaches less optimistic.

Stable semantic foundation Most approaches for aspect mining seem biased by the AspectJ style of aspect-oriented programming, where pointcuts are purely based on syntactic events. Moreover, AspectJ has been criticised recently for its lack of semantic foundation [26]. All aspect mining approaches suffer from this as they cannot base the mining on a stable semantic foundation.

To conclude, in spite of all research efforts that have been devoted to aspect mining in recent years, many problems still abound. We highlighted some of those problems and their root causes and observed that aspect mining researchers would benefit from a more stable semantic foundation; should define more precisely what exactly they are mining for and tailor their technique to that definition; and should pay more attention to how the results are represented by their technique. They would also benefit from a common framework for consistent evaluation, comparison and combination of different aspect mining techniques.

Acknowledgements

This research is partly funded by the Interuniversity Attraction Poles Programme - Belgian State - Belgian Science Policy. Andy Kellens is funded by a research grant provided by the “Institute for the Promotion of Innovation through Science and Technology in Flanders” (IWT Vlaanderen).

References

[1] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proc. Int’l Conf. Software Engineering (ICSE)*, pages 158–167, 2004.

[2] E. Baniassad, P. C. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan. Discovering early aspects. *IEEE Software*, 23(1):61–70, January-February 2006.

[3] L. Bass, M. Klein, and L. Northrop. Identifying aspects using architectural reasoning. Position paper presented at Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop at Int'l Conf. Aspect-Oriented Software Development (AOSD), 2004.

[4] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 27–36, 2005.

[5] M. Bruntink, M. DHondt, A. van Deursen, and T. Tourwé. Simple crosscutting concerns are not so simple: Analysing variability in large-scale idioms-based implementations. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 199 – 211, 2007.

[6] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 200–209, 2004.

[7] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying cross cutting concern code. *IEEE Computer Society Trans. Software Engineering*, 31(10):804–818, 2005.

[8] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231, September 2006.

[9] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 158–167, 2007.

[10] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele. Design pattern mining enhanced by machine learning. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 295–304, 2005.

[11] L. J. Fulop, T. Gyovai, and R. Ferenc. Evaluating C++ design pattern miner tools. In *Proc. Workshop Source Code Analysis and Manipulation (SCAM)*, pages 127–138, 2006.

[12] W. Griswold, Y. Kato, and J. Yuan. Aspect browser: Tool support for managing dispersed aspects. In *Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems*, 1999.

[13] Y.-G. Gueheneuc, K. Mens, and R. Wuyts. A comparative framework for design recovery tools. In *Proc. European Conf. Software Maintenance and Reengineering (CSMR)*, pages 123–134. IEEE Computer Society, 2006.

[14] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition in legacy code. In *Workshop on Advanced Separation of Concerns, Int'l Conf. Software Engineering (ICSE)*, 2001.

[15] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 19–28, 2006.

[16] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Trans. Aspect-Oriented Software Development (TAOSD)*, pages 145–164, 2007.

[17] J. Krinke. Mining execution relations for crosscutting concerns. *IET Software*, 2(2):65–78, Apr. 2008.

[18] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[19] M. Marin, L. Moonen, and A. van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 29–38, 2006.

[20] M. Marin, L. Moonen, and A. van Deursen. Soquet: Query-based documentation of crosscutting concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 758–761, 2007.

[21] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 132–141, 2004.

[22] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views — a case study. *Journal on Computer Languages, Systems and Structures*, 32(2–3):140–156, July-October 2006.

[23] M. P. Monteiro and J. M. Fernandes. Object-to-aspect refactorings for feature extraction. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, 2004.

[24] A. Rashid, P. Sawyer, A. M. D. Moreira, and J. Araújo. Early aspects: A model for aspect-oriented requirements engineering. In *Joint Int'l Conf. Requirements Engineering (RE)*, pages 199–202, 2002.

[25] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 406–416, 2002.

[26] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 481–497, 2006.

[27] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119, 1999.

[28] B. Tekinerdogan and M. Aksit. Deriving design aspects from canonical models. In S. Demeyer and J. Bosch, editors, *Workshop Reader of the 12th European Conf. Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 410–413, 1998.

[29] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proc. Working Conf. Reverse Engineering (WCRE)*, 2004.

[30] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. Workshop Source Code Analysis and Manipulation (SCAM)*, 2004.

[31] C. Zhang and H. Jacobsen. Extended aspect mining tool. http://www.eecg.utoronto.ca/~czhang/amtex, 2002.

[32] C. Zhang and H.-A. Jacobsen. PRISM is research in aspect mining. In *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 20–21, 2004.

[33] C. Zhang and H.-A. Jacobsen. Efficiently mining crosscutting concerns through random walks. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 226–238, 2007.